Universidad
Carlos III de Madrid
www.uc3m.es

Trabajo de fin de grado.

Aceleración con GPU de algoritmos de reconocimiento biométrico mediante firma manuscrita on-line.

Titulación: Grado en Ingeniería Electrónica Industrial y Automática

Autor: Alejandro García Molina

Tutor: Luis Mengíbar Pozo

Leganés, Septiembre de 2013

Índice general

1. Introducción

1.1 Descripción

El presente trabajo muestra la implementación del algoritmo DTW (Dinamic Time Warping) utilizando CUDA (Compute Unified Device Architecture). El motivo del uso de CUDA es permitir la paralelización del algoritmo que, como se verá más adelante, requiere una gran cantidad de cálculos.

Este algoritmo se ha aplicado al reconocimiento de firmas manuscritas, uno de los campos en los que se puede usar. El desarrollo del algoritmo DTW requiere una cantidad de cálculos que, como posteriormente se expondrá, evoluciona con el cuadrado del tamaño de las muestras comparadas, por lo que su aceleración podría suponer una gran ventaja.

Además del desarrollo de la implementación en paralelo, el trabajo muestra una comparación entre una implementación convencional del algoritmo, mediante computación secuencial, y el algoritmo paralelizado. De esta forma se podrá evaluar la utilidad de esta implementación.

1.2 Objetivos

Los objetivos principales del trabajo son la **implementación del algoritmo** y su **comparación** con una implementación secuencial. Para alcanzar estos objetivos, será necesario además realizar las siguientes tareas:

- **Aprendizaje del manejo de CUDA**: Se parte de un conocimiento completamente nulo de este lenguaje de programación, por lo que antes de comenzar la implementación del algoritmo será necesario estudiarlo. Esto incluye el hardware de la GPU, su funcionamiento, el lenguaje de programación y las herramientas necesarias para escribir y compilar el código.
- **Estudio del algoritmo DTW**: Qué es y por qué se utiliza. Cual es su coste computacional y en qué medida puede ser paralelizado.
- **Manejo de una base de datos de firmas**: Para evaluar el algoritmo, se dispone de una base de datos de firmas verdaderas y falsas de 100 sujetos(MCYT, se describe con detalle más adelante). Estos datos están almacenados en ficheros ".fpg", destinados a ser usados por el programa de cálculo Matlab.
- **Acondicionamiento de datos**: Es necesario obtener datos nuevos a partir de los que incluyen los archivos (por ejemplo, velocidad a partir de posiciones).

1.3 Planificación

Antes de comenzar con el trabajo, se realizó una planificación del mismo. Dado que un trabajo de fin de grado de 12 créditos ECTS debe comprender un mínimo de 300 horas de trabajo, se usó esta cifra como base de dicha planificación.

Se decidió dividir esas 300 horas en 6 meses, en bloques de 50 horas por mes. Por la situación académica del autor, el proyecto se inició en febrero, y se excluyeron de la planificación los meses de mayo y junio, por ser épocas de exámenes en la universidad. De esta forma, quedó el trabajo distribuido en dos bloques de tres meses, febrero, marzo y abril, y julio, agosto y septiembre. La fecha límite de entrega de la memoria es antes del fin de septiembre, pero al ser necesario también preparar una presentación, se decidió incluir el mes entero en la planificación.

El presente trabajo, como puede verse en el punto anterior, objetivos, presenta una carga muy grande de aprendizaje y documentación, por lo que estas partes constituirán la mayor parte de los 3 primeros meses. Existen tres áreas básicas en las que podemos dividir esta documentación: Reconocimiento de firmas, el algoritmo DTW y el lenguaje CUDA. Además de estas áreas, también se investigará sobre un campo un poco más amplio, para tener una mejor visión de ellas. Por ejemplo, además de la firma, se estudiará la identificación biométrica en general.

Dado que estas tres áreas tienen una importancia capital en el proyecto, se decidió dedicar un mes a cada una.

Una vez completada la fase de aprendizaje, queda la parte práctica. Esta consiste fundamentalmente en implementar el algoritmo DTW de forma paralela y secuencial y evaluar su rendimiento. Además, una vez finalizado todo lo anterior, será necesario redactar una memoria del trabajo realizado y elaborar una presentación.

La tarea más compleja de las anteriores será la implementación del algoritmo, a la que se decide dedicar 75 horas. 25 serán para la evaluación del rendimiento, y las 50 restantes para la redacción de la memoria y la preparación de la presentación.

A continuación, en la tabla 1.1, se muestra una planificación general de todo el trabajo, indicando las partes principales en que se dividirá cada tarea, así como los puntos más críticos.

Meses	Tarea	Subtareas	Comentarios
febrero	50 h: Biometría y reconocimiento de firmas	20 h: Documentación sobre biometría en general 20 h: Documentación sobre reconocimiento de firmas **10h: Documentación sobre la base de datos a utilizar**	Hay que destacar la importancia de analizar la base de datos que se va a utilizar en el proyecto, ya que será necesario abrirla y manejarla correctamente.
marzo	50 h: Algoritmo DTW	20 h: Documentación sobre DTW **30 h: Desarrollo de estrategias de paralelización**	Aunque es importante también estudiar el propio algoritmo DTW, se destaca el análisis de las estrategias de paralelización, ya que es en lo que se basa todo el proyecto.
abril	50 h: CUDA	25 h: Documentación sobre aspectos teóricos: hardware, paralelismo, etc. 25 h: Documentación sobre el lenguaje de programación	Toda la documentación sobre CUDA es importante, por lo que no se destaca ningún apartado en particular.
mayo y junio	Pausa por exámenes		Aunque estos dos meses se dejan vacíos, será posible emplear algo de tiempo en completar las tareas anteriores
julio	75 h: Implementación de DTW 25 h: Evaluación de rendimiento	15 h: Apertura de base de datos de firmas **35 h: Implementación del algoritmo en paralelo**	Destacamos la implementación del algoritmo en serie, que es objetivo del proyecto. Previamente será necesario haber conseguido abrir la base de datos.
agosto		25 h: implementación del algoritmo en serie **25 h: Evaluación del rendimiento**	Se destaca la evaluación del rendimiento, punto en el que podremos determinar si el resultado obtenido es positivo.
septiembre	50 h: redacción de la memoria y la presencación.	40 h: Redacción de la memoria 10 h: Preparación de la presentación	Tanto la memoria como la presentación son importantes, no se destaca ninguna por encima de la otra

tabla 1.1: Planificación del trabajo. Tareas clave destacadas en **negrita.**

1.4 Sobre el marco regulador

El presente trabajo no aborda ningún tema que esté sujeto a algún tipo de legislación o normativa particular.

2. Estado del arte

En el presente apartado, se presentarán los conceptos necesarios tanto para entender el trabajo en sí, como para hacerse una idea general de su ámbito de aplicación.

Se tratará la identificación biométrica, incluyendo las diferentes técnicas de identificación existentes, con especial atención en la firma manuscrita, objeto de este trabajo. También se analizará el algoritmo DTW, método empleado para la comparación de las firmas. Por último, y dado que el trabajo trata sobre la paralelización del algoritmo, se estudiarán las distintas formas de paralelización existentes en informática, de nuevo mostrando especial interés en la paralelización mediante CUDA.

2.1 Identificación biométrica

La identificación biométrica (o biometría) es la identificación de seres humanos mediante sus rasgos físicos o de conducta. La identificación biométrica es usada para la verificar la identidad de personas y el control de accesos.

Los rasgos que evalúa la biometría se denominan identificadores biométricos. Ejemplos de identificadores físicos son el iris del ojo, las huellas dactilares o el ADN. Como identificadores conductuales podemos citar la voz, la marcha o el que ocupa este trabajo, la firma.[1]

Dado que los identificadores biométricos son inherentes a cada persona, resultan más fiables que la identificación mediante objetos (como una llave o una tarjeta de acceso) o mediante conocimientos (contraseñas, datos personales, etc.)[1]. Sin embargo, el uso de la identificación biométrica presenta problemas que no tienen métodos más tradicionales, como veremos después.

2.1.2 Métodos actuales de identificación biométrica

Como ya se ha visto antes, existen dos tipos de identificadores, físicos y conductuales. A continuación se presentan los más destacados de cada grupo. La información aquí presentada puede ampliarse en [2].

2.1.2.1 Identificadores físicos

- **Huella dactilar:** Sin duda alguna, el sistema biométrico más difundido. Actualmente se emplea en control de accesos y como medio de identificación de personas, tanto mediante los dedos de esas personas como

mediante las huellas dejadas por estos en una superficie. En la figura 2.1.1 se puede ver una aplicación práctica de identificación biométrica mediante huella dactilar: desbloquear un teléfono móvil leyendo la huella del propietario.

figura 2.1.1: Ejemplo de sensor de huella dactilar en un teléfono móvil de última generación.[6]

- **Iris del ojo:** Utiliza una imagen en alta resolución del iris de una persona. Posee una tasa de errores muy baja, el identificador se encuentra relativamente protegido y es inalterable por el tiempo. Otros sistemas emplean un método similar pero usando la **retina**. En la figura 2.1.2, se puede observar un sistema de identificación de iris, empleado por el ejército de EEUU.

figura 2.1.2: Equipo de identificación biométrica mediante reconocimiento de iris.[7]

- **Identificación vascular:** Compara los vasos sanguíneos superficiales de una parte del cuerpo, como un dedo o la palma de la mano. Es un sistema no muy extendido y aún en desarrollo.[3]
- **Geometría de la mano:** Se trata de una técnica muy antigua, en la que no se han producido avances significativos desde el siglo 19, aunque algunos expertos la consideran válida. Se basa en analizar determinados elementos de la mano: grosor de las falanges, desalineación de las articulaciones, tamaño y forma de la mano en general. En la figura 2.1.3 se ve un ejemplo de aplicación de esta tecnología, en un lector para controlar la apertura de una puerta, combinado con la introducción de una clave.

figura 2.1.3: Ejemplo de lector de manos, para apertura de puertas.[8]

2.1.2.2 Identificadores conductuales

- **Voz:** La voz de una persona depende tanto de sus características fisiológicas como de sus hábitos lingüísticos.
- **Firma:** Utilizada como medio tradicional de identificación en documentos escritos, la firma constituye un identificador biométrico fiable, con gran aceptación social.
- **Escritura manuscrita:** Esta técnica se lleva empleando siglos para verificar la autenticidad de documentos. Consiste en analizar ciertos rasgos característicos de la escritura de una persona, mediante un documento auténtico, para compararlos con otro documento cuya autoría se quiere comprobar.
- **Dinámica de tecleo:** Se basa en analizar los tiempos entre pulsaciones, así como los tiempos de las pulsaciones propiamente dichas, elaborando un patrón que permita reconocer a la persona que teclea.

2.1.3 Medida del rendimiento

Para evaluar el rendimiento de un sistema de identificación biométrico, es necesario tener en cuenta los distintos errores que pueden producirse. A cada uno

de ellos se le asocia un ratio, correspondiente al porcentaje de casos en los que sucede. Se citan a continuación los principales, pudiendo profundizar en el tema en [4]:

- **FAR**(false accept rate)**:** Este error consiste en dar por buena una entrada falsa. Aumenta a medida que se rebajan los requisitos para dar una identificación por buena.
- **FRR**(false reject rate)**:** El caso contrario al anterior. No se acepta una entrada correcta. Aumenta al aplicar requisitos más elevados.
- **FER**(failure to enroll rate)**:** Fallo al obtener un modelo numérico de una entrada válida. Se puede dar, por ejemplo, por entradas de baja calidad.
- **FTC**(failure to capture ratio): Aplicable a sistemas automáticos. Probabilidad de que el sistema no sea capaz de capturar una entrada, si esta se realiza correctamente.

De los dos primeros, FAR y FRR, se observa una relación: ambos dependen de la tolerancia que apliquemos en la identificación. Modificando esta, es posible mejorar uno de estos errores a costa de aumentar el otro. Por este motivo, no resultan fiables por separado para medir el rendimiento. Para evaluar el rendimiento de forma objetiva, se emplea el **EER(equal error rate)**, que corresponde al valor del FAR y el FRR para un valor de sensibilidad que los haga iguales. A continuación, en la figura 2.1.4, podemos ver la evolución del FAR y el FRR, en función de la sensibilidad que se aplique, en el punto en el que ambos se igualan obtenemos el EER.

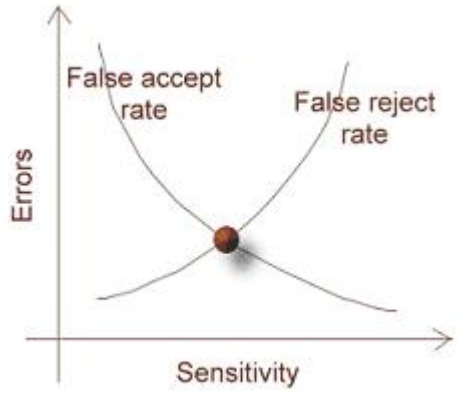

figura 2.1.4: Evolución del FAR y el FRR con la sensibilidad. El EER corresponde al punto de unión de ambas curvas.[9]

2.1.4 Inconvenientes de la identificación biométrica

La identificación biométrica no está exenta de problemas. Algunos suponen una cuestión técnica, como determinar la voluntariedad de la identificación, mientras que otros presentan debates éticos, como los relacionados con la privacidad. A continuación introducimos los más destacables, pudiendo ampliar esta información en [5].

2.1.4.1 Privacidad

En ocasiones, la información necesaria para la identificación biométrica contiene mucha mas información que la mera identidad de la persona. Un claro ejemplo es el ADN, del cual, además de una identificación única, pueden obtenerse resultados de pruebas médicas, como predisposición a enfermedades.

Otro problema es la posible identificación involuntaria de un sujeto. Si el rasgo usado para la identificación está a plena vista, puede ocasionar que una persona sea identificada sin su conocimiento, lo que atenta contra su intimidad. Por otra parte, algunos métodos no pueden verificar la voluntariedad de la identificación. Por ejemplo, se podría poner el dedo de una persona incapacitada sobre un escáner de huellas y obtener una identificación positiva.

2.1.4.2 Riesgo físico para los usuarios

Utilizar nuestro cuerpo a modo de llave conlleva el riesgo de que alguien trate de arrebatarnos esa llave. Cuando una identificación biométrica protege algo de gran valor, la integridad física de quien tenga acceso puede verse amenazada en mayor grado que si la identificación fuese mediante un método mas tradicional, como una llave.

2.1.4.3 Identificadores irreemplazables

Los rasgos que emplea la identificación biométrica son únicos. No existe forma de obtener un duplicado, por este motivo son tan seguros. Como contrapartida tenemos el hecho de que si ese rasgo se altera, la identificación será imposible. Ciertos rasgos son más vulnerables que otros a ser modificados. Las huellas dactilares, por ejemplo, pueden verse alteradas por lesiones en la piel de los dedos.

2.1.4.4 Coste computacional

Otro inconveniente de los sistemas de identificación biométrica es lo complejo que resulta ejecutarlos para una máquina. Requieren técnicas mucho más complejas que por ejemplo la lectura de una contraseña. Este inconveniente, sin embargo, es puramente técnico, y con el desarrollo de nuevas técnicas de comparación y el incremento de potencia de los equipos electrónicos irá disminuyendo progresivamente.

Referencias

[1]Marino Tapiador Mateos, Juan A. Sigüenza Pizarro, "Introducción a la biometría" en: "Tecnologías biométricas aplicadas a la seguridad" pags: 3-19.

[2]Marino Tapiador Mateos, Juan A. Sigüenza Pizarro, "Biometría estática" y "Biometría dinámica"en: "Tecnologías biométricas aplicadas a la seguridad" pags: 81-264.

[3] Stan Z. Li, "Back-of-Hand Vascular Pattern Recognition" en: "Encyclopedia of Biometrics" pags: 55-59

[4] Marino Tapiador Mateos, Juan A. Sigüenza Pizarro, "Medición del rendimiento" en: "Tecnologías biométricas aplicadas a la seguridad" pags: 64-71.

[5] Stan Z. Li, "Privacy" y "User Aceptance" en: "Encyclopedia of Biometrics" pags: 1092-1098 y 1357-1362.

[6] www.apple.com

[7]http://commons.wikimedia.org/wiki/File:USMC_Sergeant_identifies_Baghdaddi_city_council_member_with_iris_scanner.jpg

[8] Stan Z. Li, "Hand geometry" en: "Encyclopedia of Biometrics" pag: 678.

[9] http://commons.wikimedia.org/wiki/File:Biometrics_error.jpg

2.2 Reconocimiento de firmas

Una vez revisados los distintos medios utilizados en la actualidad para la identificación biométrica, Se va a profundizar en el reconocimiento de firmas, tema sobre el que trata este trabajo. Se analizarán las dos grandes ramas de esta técnica: reconocimiento on-line y reconocimiento off-line. Dado que este trabajo trata sobre el reconocimiento de firmas on-line, este será abordado con más detalle. Para ampliar la información aquí presentada, se puede consultar [1].

2.2.1 Ventajas de la firma como medio de identificación

El empleo de la firma como medio de identificación ofrece varias ventajas frente a otros tipos de identificación biométrica:

- **Facilidad de ejecución:** El hecho de firmar algo es algo que todos hemos realizado en múltiples ocasiones. Resulta un gesto natural, no es necesario ningún instrumento extraño, ni instrucciones concretas, mas allá de un "firme aquí".
- **Facilidad de verificación manual:** Salvo en el caso de falsificadores entrenados, basta una inspección visual rápida.
- **Aceptación social:** La firma es un medio tradicional de identificación en todo el mundo. Es la forma en la que desde hace siglos una persona demuestra su autoría o conformidad con un escrito. Por este motivo, el pedir a alguien que firme, no le supone un rechazo, como el que podría causarle el analizar sus ojos, por ejemplo.

Estas ventajas hacen de la firma un identificador biométrico interesante a la hora de identificar personas.

2.2.2 Métodos de adquisición de datos: on-line y off-line

Es posible adquirir los datos de una firma de dos maneras: la primera es tomando una firma ya escrita y extrayendo de ella todo lo que sea posible. A este primer método se le denomina adquisición off-line, ya que el momento de la ejecución de la firma y el de captura de datos difieren en el tiempo. El segundo, consiste en analizar la firma mientras esta es ejecutada, empleando para ello medios que permitan capturar en tiempo real la máxima información posible, como tabletas digitalizadoras, o dispositivos de pantalla táctil. A este método, en el que coinciden ejecución y captura, se le denomina adquisición on-line.

La captura off-line permite una adquisición de datos más limitada: un grafólogo experto puede analizar una firma manuscrita y obtener datos sobre presión, o velocidad, pero en el caso de que queramos automatizar el proceso, estos datos son tan difíciles de evaluar, que se consideran perdidos. Por tanto, consideraremos como única información de la captura off-line, la impresión de la firma.

Por el contrario, la captura on-line ofrece mayores posibilidades: dado que monitorizamos el lápiz mientras se mueve, es posible determinar su velocidad. Así mismo, sistemas más avanzados ofrecen información sobre la presión ejercida por el lápiz, e incluso el ángulo con el que este escribe.

A continuación, en la figura 2.2.1, aparece un medio combinado de captura on-line y off-line: se emplea una tableta digitalizadora, con un lápiz especial, capaz de capturar presión y ángulo y además dotado de una punta de bolígrafo convencional. Sobre la tableta se coloca una hoja de papel, sobre la que queda escrita la firma. En el momento de la realización de la firma, la tableta la captura on-line. Posteriormente, la hoja es escaneada, para obtener la captura off-line.

figura 2.2.1: Tableta digitalizadora cubierta con una hoja de papel, para capturar una firma on-line y off-line simultáneamente.[3]

2.2.3 Acondicionamiento de los datos adquiridos

Como paso previo a la comparación, es recomendable acondicionar los datos tomados, para eliminar aquellas variaciones, errores o información irrelevante que dificulten la comparación. Dado que existen dos formas de capturar los datos, cada uno con unos medios distintos, el proceso de acondicionamiento cambia. Hablaremos pues por separado de ambos.

En el caso de la firma off-line, se parte de una imagen escaneada. La única información que nos es útil es la geometría de la firma, de modo que hemos de eliminar todo lo accesorio.

No nos interesan los colores ni las tonalidades que pudiera haber en la imagen, de forma que si existen, se aplicarán filtros para pasarla a blanco y negro(binarización). Solo queremos la geometría de la firma, de modo que cualquier zona en blanco, o con otros elementos puede ser recortada. Por último, es conveniente eliminar el ruido de la imagen obtenida, de forma que, por ejemplo, se rellenen huecos en las líneas.

Otros acondicionamientos posibles, según nuestras necesidades, pueden ser escalar la firma a un tamaño estándar, dividirla en celdas más pequeñas, o realizar transformaciones especiales, destinadas a facilitar la comparación, como rellenar curvas cerradas, o unir trazos que estén próximos mediante líneas horizontales.

Si empleamos la captura on-line, partimos con mucha ventaja: La geometría de la firma que capturamos ya está binarizada y no tiene ruido. En el caso anterior, "recortábamos" la firma de la imagen en que se encontraba. En el caso de la firma dinámica, como la imagen ya está en unas coordenadas x e y, lo que hemos de hacer es normalizar esas coordenadas, a fin de que todas las firmas tengan un origen de coordenadas similar. Es frecuente utilizar el primer punto capturado, o el centro de masas de la firma. También es posible ajustar el tamaño de la firma a una escala determinada, o rotarla para eliminar cualquier posible inclinación. Las características adicionales que obtenemos no necesitan preparación, mas allá de un reescalado si comparamos firmas obtenidas con instrumentos de sensibilidad distinta(Por ejemplo, un lápiz con 1024 niveles de presión frente a uno con 2048).

2.2.4 Características a comparar

Una vez tenemos los datos, se ha de determinar cuales son las características más discriminatorias a la hora de comparar dos firmas. Una característica ideal es aquella que para un mismo usuario presenta variaciones pequeñas, pero al analizar falsificaciones, encontramos variaciones grandes.

Antes de proseguir, conviene señalar los dos casos de falsificador que pueden encontrarse:

- **Falsificador real**: Es una persona que, conociendo la firma original, trata de imitarla para pasar la prueba.
- **Falsificador casual:** Es simplemente una persona que ejecuta su propia firma.

Si se tiene un sistema en el que la firma sirve, por ejemplo, para controlar accesos de un gran número de personas, seguramente sea necesario preocuparse por el segundo tipo: personas que tratan de acceder a algo a lo que no tienen acceso, pero de forma casual, puede que incluso por desconocimiento. Si en cambio, un reconocimiento de firma protege algo de gran valor, es más probable tener falsificadores del primer tipo, expertos en imitar firmas.

En el caso de la firma off-line, no hay posibilidad de elección: solo se dispone de la geometría de la firma. Este rasgo es el más fácil de imitar para un falsificador real, por lo que el uso de la firma off-line para evitar este tipo de falsificación está muy limitado.

En el caso de la firma on-line, no obstante se puede disponer de un buen número de características. Según estudios realizados [2] se pueden definir dos grupos de características idóneas, en función del escenario en que nos encontremos:

- **Escenario casual:** Se busca una baja tasa de falsos negativos, más que una seguridad estricta. La combinación más adecuada es la posición y la velocidad en los dos ejes, esto es: **(x, y, dx, dy).**
- **Escenario seguro:** conlleva una mayor dificultad para el falsificador. Para ello, se introduce la presión como elemento adicional, descartando la coordenada x. Queda entonces: **(y, dx, dy, p).**

El conjunto para escenarios casuales presenta la ventaja de que no es necesario capturar la presión, por lo que pueden emplearse dispositivos más sencillos.

2.2.5 Métodos de comparación de firmas

Los múltiples métodos de comparación de firmas que existen actualmente, pueden ser agrupados en tres grandes bloques: alineamiento de características, sistemas de aprendizaje automático y modelado estadístico.

- **Alineamiento de características**: se basa en deformar una muestra hasta que coincide con la otra. Cuantificando esa deformación, puede saberse cómo de parecida es una firma a la otra. Dos firmas genuinas presentarán variaciones pequeñas, por lo que su deformación será pequeña también. Uno de los métodos más utilizados dentro del alineamiento de características es el algoritmo DTW(Dynamic Time Warping). DTW emplea un vector para cada serie de datos, donde cada elemento contiene los datos que compararemos (por ejemplo, el elemento 'i' del vector contiene el i-ésimo valor de la posición, velocidad y presión). Analizando las distancias entre los elementos de ambos vectores, el algoritmo obtiene una medición de la similitud entre ellos.
- **Técnicas basadas en sistemas de aprendizaje automático**: se basan en un sistema capaz de aprender, mediante ejemplos o ensayo y error, a distinguir entre firmas verdaderas y falsas.
- **Técnicas de modelado estadístico**: analizan un grupo de firmas auténticas, obteniendo su variabilidad. Posteriormente, a la hora de comparar una firma, se verifica si encaja dentro de ese modelo estadístico.

Dado que en este trabajo se ha empleado la técnica de alineamiento de características mediante el algoritmo DTW, se explicará este algoritmo con más detalle en el apartado homónimo.

Referencias

[1] Marino Tapiador Mateos, Juan A. Sigüenza Pizarro, "Reconocimiento de la firma escrita" en: "Tecnologías biométricas aplicadas a la seguridad" pags: 201-222.

[2] Juan Manuel Pascual Gaspar "Evaluación conjunta" en: Uso de la Firma Manuscrita Dinámica para el Reconocimiento Biométrico de Personas en Escenarios Prácticos pags: 66-68.

[3] Stan Z. Li, Dynamic signature databases, en: Encyclopedia of Biometrics, pag. 1184.

2.3 Bases de datos de firmas On-line

Hasta hace poco, la investigación sobre reconocimiento de firmas se hacía mediante bases de datos privadas. Esto ha limitado la comparación de resultados entre distintos sistemas de reconocimiento, además de afectar a la validez de los resultados, si las muestras no eran lo suficientemente grandes.

En los últimos años han aparecido diversas bases de datos públicas. Muchas de estas bases de datos están disponibles de forma gratuita. En ocasiones no sólo contienen firmas, sino también otros identificadores biométricos, como huellas dactilares, obtenidos a la par que las firmas.

La forma más habitual de obtener los datos es una tableta digitalizadora, que proporciona datos de posición, presión e inclinación del lápiz. En otros casos, se usó una pantalla táctil como la de una agenda electrónica (PDA), que solo proporcionó información de la posición del lápiz. En algunos casos se colocó una hoja de papel sobre la tableta, obteniendo una impresión de la firma que posteriormente fue escaneada para usarse en comparaciones off-line.

A diferencia de otros identificadores biométricos, en el caso de las firmas no solo es necesario distinguir entre distintos individuos, también es necesario descartar las falsificaciones. Por este motivo, junto a las firmas auténticas se incluye un conjunto de firmas falsas. [1]

2.3.2 Bases de datos más utilizadas

A continuación se exponen las principales bases de datos de firmas para reconocimiento dinámico. Se hará hincapié en el número de firmas verdaderas y falsas, así como su forma de obtención. Esta información puede ampliarse en [1].

- **Base de datos PHILIPS:** Consta de 51 individuos, con 30 firmas auténticas por usuario y más de 3000 falsas, sin tener un número fijo por individuo. Presenta firmas falsas hechas por sujetos corrientes con y sin practicar previamente y también algunas hechas por grafólogos forenses. Obtenida mediante tableta digitalizadora, a una frecuencia de 200Hz, ofrece posición, presión e inclinación.
- **Base de datos MCYT[2]:** De las más amplias, con 330 individuos, cada uno con 25 firmas genuinas y 25 falsas. También es de las más populares, siendo usada por más de 50 grupos de investigación en todo el mundo. Los individuos realizaban 5 firmas propias, después imitaban 5 de otros usuarios, repitiendo el proceso 5 veces. Obtenida mediante tableta digitalizadora, a una frecuencia de 100Hz, ofrece posición, presión e inclinación. También ofrece las firmas de 75 usuarios en formato off-line.
- **Base de datos BIOMET:** 84 individuos, con 15 firmas auténticas y hasta 12 falsas. Las firmas se tomaron en dos sesiones, separadas entre 3 y 5 meses. Algunos sujetos no completaron todas las firmas, por lo que faltan 8 originales y 54 falsas. Obtenida mediante tableta digitalizadora, a una frecuencia de 100Hz, ofrece posición, presión e inclinación.

- **Base de datos SVC2004:** En realidad son dos bases de datos, usadas en la Signature Verification Competition (SVC) de 2004. Los datos de que disponen se corresponden con los requisitos del concurso: una solo presenta datos de posición, mientras que la otra también incluye presión e inclinación. Cada una tiene 40 individuos, con 20 firmas genuinas y 20 falsas cada uno. Presenta firmas occidentales y asiáticas. Un dato relevante de esta base de datos es que se pidió a los individuos que no usaran su firma real, para preservar su privacidad. En su lugar, se les pidió que inventaran y ensayaran una firma ficticia.
- **Base de datos SUSIG:** Consta de dos grupos: uno en el que los usuarios no veían lo que escribían y otro en el que se mostraba en una pantalla lo que la tableta capturaba. El segundo grupo se capturó 4 años después del primero, sin que coincidan los sujetos de ambos. Cada grupo tiene 100 usuarios, con 10 firmas auténticas y 10 falsas. La primera se capturó en una sesión, la segunda en dos.
- **Base de datos MyIDea:** Consta de 70 individuos, con 18 firmas genuinas y 18 falsas, realizadas en tres sesiones. Se obligó a los individuos a repetir en voz alta lo que estaban escribiendo.
- **Base de datos BiosecurID:** Hecha por 6 instituciones españolas, se realizó en 4 sesiones repartidas en 4 meses, con 4 firmas en cada una. Además, a cada usuario se le pidió que falsificara tres firmas por sesión, primero sólo con una imagen como referencia, y más adelante con más ayudas, como indicaciones de la presión correcta de cada trazo. También se almacenó información personal de los sujetos, como género, edad, zurdera, uso de ayudas visuales, etc. con el objetivo de permitir investigaciones sobre grupos concretos. Obtenida mediante tableta digitalizadora, a una frecuencia de 100Hz, ofrece posición, presión e inclinación. También dispone de versión off-line de las firmas.
- **Base de datos BioSecure:** Consta de tres grupos: uno recogido a través de internet, otro elaborado mediante una tablera digitalizadora y un tercero mediante una PDA. Los dos últimos (llamados DS2 y DS3 respectivamente) fueron realizados por los mismos 667 individuos. Las firmas se tomaron en dos sesiones, separadas por unos dos meses. En cada sesión se capturaron 3 grupos de 5 firmas genuinas, y entre cada grupo se pidió a los participantes que imitaran 5 firmas. Para recrear la peor situación posible, se les ofreció toda la información posible: imagen de la firma original, diagramas con la presión y los trazos originales, incluso en el grupo de la PDA, a algunos se les mostraba la firma original para que escribieran sobre ella. Estas facilidades, junto a la ausencia de presión e inclinación, hacen del grupo DS3 un reto muy difícil para los sistemas de reconocimiento.

A continuación, en la **figura 2.3.1**, Se muestra una tabla con un resumen de todas las bases de datos expuestas:

Name	Device	Users	Sessions	Signatures per user		Signals	Interval between sessions
				Genuine	Forgeries		
PHILIPS	Pen tablet	51	3–5	30	up to 70	x,y,p,θ,γ	1 week approx.
BIOMET	Pen tablet	84	3	15	up to 12	x,y,p,θ,γ	3–5 months
MCYT	Pen tablet	330	1	25	25	x,y,p,θ,γ	-
SVC2004 Task 1	Pen tablet	40	2	20	20	x,y	min. 1 week
SVC2004 Task 2	Pen tablet	40	2	20	20	x,y,p,θ,γ	min. 1 week
SUSIG Blind Subcorpus	Pen tablet	100	1	8 or 10	10	x,y,p	-
SUSIG Visual Subcorpus	Pen tablet	100	2	20	10	x,y,p	1 week approx.
MyIDea	Pen tablet	ca. 100	3	18	18	x,y,p,θ,γ	days to months
BioSecurID	Pen tablet	400	4	16	16	x,y,p,θ,γ	1 month approx.
BioSecure DS2	Pen tablet	ca. 650	2	30	20	x,y,p,θ,γ	1 month approx.
BioSecure DS3	PDA	ca. 650	2	30	20	x,y,p,θ,γ	1 month approx.

figura 2.3.1: Comparativa de bases de datos [1].

2.3.3 Base de datos MCYT

Para este proyecto empleamos la base de datos MCYT, por lo que se entrará un poco más en detalle en sus características [2].

Ya se mencionó que esta base de datos se compone de 330 individuos, con 25 firmas originales y 25 falsas por cada uno. Sin embargo, la versión de que se dispone es la lanzada en 2004, que cuenta con 100 individuos.

Las firmas fueron capturadas mediante una tableta digitalizadora WACOM INTUOS A6. Esta tableta cuenta con una superficie útil de 127 x 97 mm, correspondiente a una hoja de papel tamaño DIN-A6 (la cuarta parte de un DIN-A4). En términos de posición, posee una resolución de 100 puntos/mm, en ambos ejes de coordenadas, lo que da un total de 12.700 x 9.700 puntos para toda su superficie. Su precisión se sitúa en +/- 0.25mm. Además de posición, es capaz de medir la presión del lápiz, en 1024 niveles, su inclinación, el ángulo que forma con la tableta, y el azimut, el ángulo que forma con respecto al eje X de la tableta. En ambos ángulos, la sensibilidad es de 0,1°. En la **figura 2.3.2**, vemos un diagrama de los ángulos capturados por la tableta. Además, esta tableta es capaz de detectar el lápiz cuando se encuentra a menos de 10mm de su superficie, de modo que incluso en los momentos en que se despega del papel, los datos pueden seguir recopilándose.

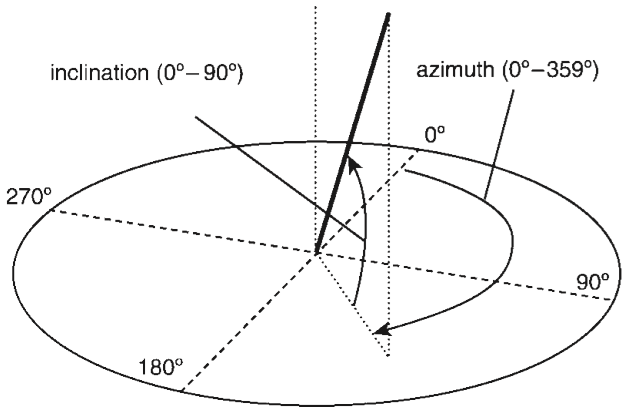

figura 2.3.2:Ángulos capturados por la tableta.[2]

Los rangos de las medidas son: Para el eje X, [0-12700], eje Y, [0-9700], presión[0-1024], azimut,[0-3600] e inclinación, [300-900]. Nótese en este último que la escala empieza en 300, esto es debido a que el ángulo mínimo con el que el lápiz puede escribir es 30° con respecto a la horizontal. La frecuencia a la que se toman estos datos es de 100 capturas por segundo.

Para recoger las firmas, se pidió a cada usuario que hiciera 5 grupos de 5 firmas. Tras cada grupo, se le pidió que imitara 5 veces la firma de una de las 5 personas anteriores. De este modo, al finalizar, había realizado 25 originales y 5 falsas de cada uno de los 5 sujetos anteriores.

Para realizar cada grupo de falsificaciones, se pidió al individuo que observase una firma original, y tratara de imitarla al menos 10 veces antes de proceder a capturar las falsificaciones válidas. Además, se les pidió que escribieran de forma natural, sin interrupciones bruscas, ni a velocidades demasiado lentas.

Referencias

[1] Stan Z. Li, Dynamic signature databases, en: Encyclopedia of Biometrics, pags. 1179-1184

[2] J. Ortega-Garcia, J. Fierrez-Aguilar, D. Simon, J. Gonzalez, M. Faundez-Zanuy, V. Espinosa, A. Satue, I. Hernaez, J.-J. Igarza, C. Vivaracho, D. Escudero and Q.-I. Moro “MCYT baseline corpus: a bimodal biometric database”, IEE Proc.-Vis. Image Signal Process., Vol. 150, No. 6, December 2003, pags. 395-401.

2.4 El algoritmo DTW

El algoritmo DTW es empleado en multitud de campos para comparar series de datos. Tiene como gran virtud su capacidad para reconocer similitudes entre dos series de datos aunque se encuentren desplazadas o desfasadas.[1]

Su funcionamiento se basa en encontrar la distancia más corta que separa los puntos de dos series de datos, como podemos ver en la **figura 2.4.1**, el algoritmo trata buscar los puntos de ambas series que son similares, aunque no estén alineadas:

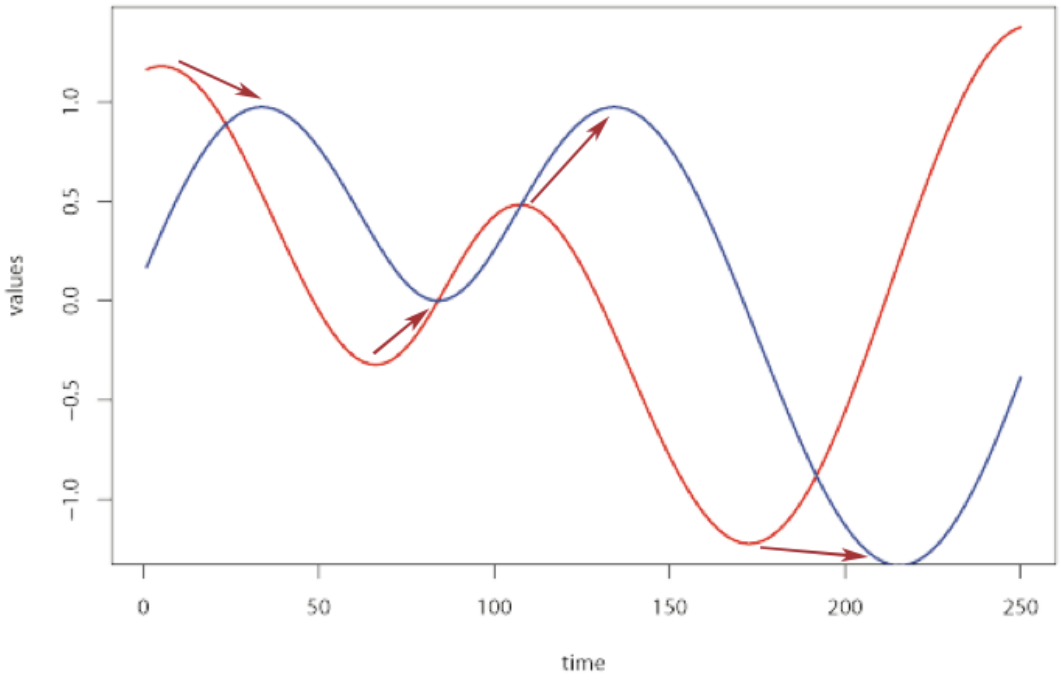

figura 2.4.1: Dos series temporales desplazadas. Las flechas muestran las relaciones que busca el algoritmo DTW [1].

Uno de los primeros usos de este algoritmo fue el reconocimiento del habla[2], si bien en la actualidad se emplea en multitud de áreas: reconocimiento de lenguaje de signos, búsquedas en bases de datos de series temporales, alineado de secuencias de proteínas o el que ocupa el presente trabajo: reconocimiento de firmas.[1]

2.4.2 Desarrollo del algoritmo y coste computacional

Para comparar dos series de datos, que llamaremos A_n y B_m, compuestas cada una por n y m puntos respectivamente, comenzamos calculando la distancia entre cada

uno de los punto de una de ellas con todos los puntos de la otra. Estas distancias, denominadas costes locales, se almacenan en la matriz de costes locales, C_{nxm}. Cada elemento de C puede definirse por tanto como muestra la **figura 2.4.2**:

$$c_{i,j} = dist(a_i, b_j)$$

ecuación 2.4.2: Coste local entre los elementos i y j de A y B respectivamente

Esta distancia debe calcularse para cada par de puntos de A y B, por tanto, **son necesarios n·m cálculos.**

Una vez obtenida C, si analizamos los datos que contiene, veremos que hay zonas en las que los costes son bajos, correspondientes a puntos de ambas series que tienen valores similares, y zonas de costes muy altos, que representan partes de las series que son muy diferentes entre sí. Para comparar la similitud entre ambas series, tenemos que buscar el camino que recorre la matriz C desde el punto 1,1 hasta el punto n,m, que pasa por las zonas de menor coste posible.

En la **figura 2.4.3** vemos una representación gráfica de la matriz C, donde el color verde representa un coste bajo y el amarillo costes altos. Destacado en azul vemos el camino de menor coste.

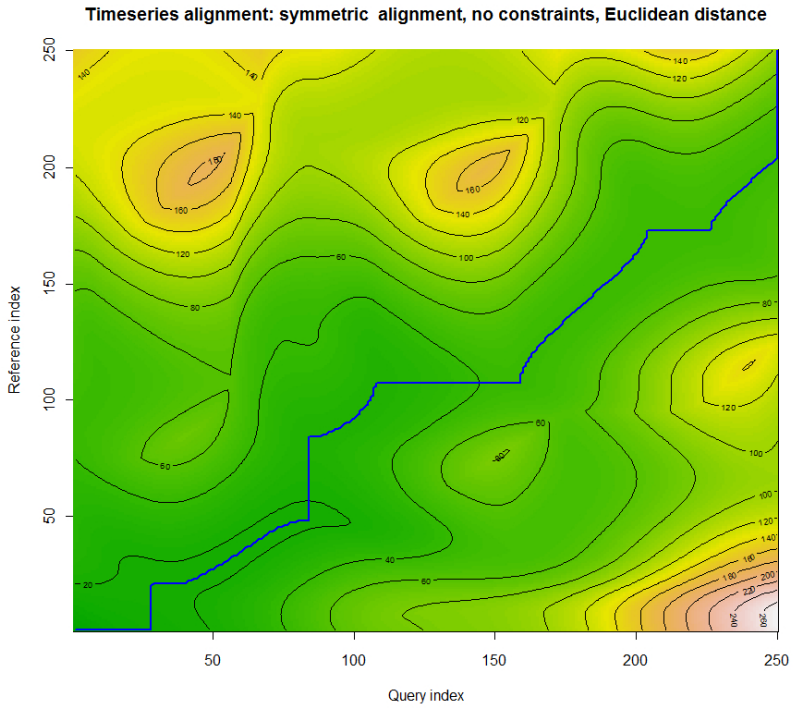

figura 2.4.3: Matriz de costes locales representada gráficamente, mostrando el camino de coste mínimo [1].

Para obtener el camino, primero debemos decidir la forma en la que nos vamos a desplazar por la matriz. La condición básica es que el camino nunca debe volver atrás. Por tanto, siempre debemos desplazarnos en una dirección que haga que i y j

crezcan, o al menos uno de ellos. Dicho de otro modo, dado un elemento cualquiera del camino (i,j), el elemento siguiente debe ser (i,j+1), (i+1,j) o (i+1,j+1).

Para obtener el camino de menor coste, la única alternativa es calcularlos todos. La forma más eficiente de hacer esto es calcular la matriz de costes totales, D_{nxm}, en la que cada elemento representa el coste del camino de menor coste que llega hasta él. La ventaja de esta matriz, es que cada elemento puede escribirse como el coste local de ese punto, más el menor de los costes acumulados de los elementos anteriores. Teniendo en cuenta esto, y la condición de que el camino siempre debe avanzar hacia delante, podemos escribir el valor de cualquier elemento de D de la siguiente manera:

$$d_{i,j} = c_{i,j} + \min\left(d_{i-1,j}, d_{i,j-1}, d_{i-1,j-1}\right)$$

ecuación 2.4.4: Expresión de un punto genérico de D.

Esta expresión general cuenta con algunas excepciones:

- El primer punto, $d_{1,1}$, queda igual a $c_{1,1}$, ya que no existe ningún punto anterior cuyo coste sumarle.
- En la primera columna de puntos, el camino sólo puede desplazarse en vertical, ya que no hay puntos anteriores en horizontal y el camino sólo avanza hacia delante. La expresión de cualquier punto de esta columna puede expresarse entonces simplemente como la suma de todos los elementos desde ese punto hasta el origen, ambos incluidos:

$$d_{1,j} = \sum_{k=1}^{j} c_{1,k}$$

ecuación 2.4.5: Expresión de un punto genérico de la primera columna de D.

- De forma análoga al caso anterior, la primera fila presenta un caso similar: el camino sólo puede desplazarse horizontalmente, por tanto la expresión de sus puntos queda:

$$d_{i,1} = \sum_{k=1}^{i} c_{k,1}$$

ecuación 2.4.6: Expresión de un punto genérico de la primera fila de D.

Existen otras formas de obtener estas distancias acumuladas, variando el denominado "paso", esto es, los elementos anteriores cuyos costes podemos tomar para calcular el coste del elemento actual. Dado que el objetivo de este trabajo es la evaluación del coste computacional del algoritmo y no el estudio de los distintos tipos de paso, que pueden afectar al coste acumulado, pero no al tiempo de ejecución, no se ha profundizado en este aspecto. Respecto a los distintos tipos de pasos que habitualmente se emplean, puede consultarse en [2] más información al respecto.

Para obtener la matriz D completa, es necesario calcular el coste acumulado de todos los puntos, es decir **n·m cálculos.**

Una vez completada la matriz D, en su último elemento D[n,m], obtenemos el coste total de igualar ambas series. Evaluando este coste, podemos determinar si ambas series son lo suficientemente parecidas.

Si sumamos el coste computacional de ambas etapas, obtenemos **un total de 2·n·m cálculos para completar el algoritmo.**

Si asumimos que ambas series serán de un tamaño parecido, podemos concluir de forma simplificada que el coste computacional del algoritmo crece con el cuadrado del tamaño de las series de datos.

Referencias

[1] Dynamic Time Warping Algorithm Review, Pavel Senin, Information and Computer Science Department, University of Hawaii at Manoa, Honolulu, USA, diciembre de 2008

[2] Sakoe, Shiba, "Dynamic Programming Algorithm Optimization for Spoken Word Recognition" en: "IEEE transactions on acoustics, speech, and signal processing, vol. Assp-26, no. 1, febrero 1978"

2.5 Computación paralela

El desarrollo de este proyecto se basa en la paralelización de un algoritmo para reducir su duración, antes de centrarnos en una opción concreta, revisaremos los distintos tipos de paralelismo que podemos encontrar en informática.

2.5.1 Concepto de computación paralela

Tradicionalmente, un programa informático se desarrolla de forma secuencial: Una tarea se divide en instrucciones que son ejecutadas una a una por un procesador. La computación paralela consiste en ejecutar esas instrucciones de forma simultánea, en varios procesadores[1]. Más adelante veremos que también se pueden paralelizar instrucciones con un solo procesador.

Como ventaja principal de la computación paralela frente a la secuencial podemos señalar la mejor relación entre velocidad y consumo: El tiempo de ejecución de una instrucción varía linealmente con la frecuencia del procesador, sin embargo, el consumo eléctrico de ese procesador varía linealmente con el cuadrado de la frecuencia[2]. Por este motivo, es más rentable usar varios procesadores a una velocidad menor que uno muy rápido.

Los inconvenientes principales de la computación paralela son:

- **La mayor dificultad a la hora de escribir los programas:** La paralelización debe implementarse explícitamente, y contrasta con la manera en que una persona piensa: de forma secuencial.
- **Las dependencias entre instrucciones:** Si una instrucción necesita el resultado de otra para ejecutarse, no se puede ejecutar ambas a la vez.

2.5.2 Limitaciones de la computación paralela

En 1967, Gene Amdahl publicó un artículo sobre las limitaciones de la computación paralela [3]. La conclusión más importante de este artículo es que existe un límite en el ahorro de tiempo que produce la paralelización, y ese límite lo provoca la parte de la tarea que debe ejecutarse de forma secuencial. De este artículo se extrajo la conocida como "Ley de Amdahl", que puede escribirse así:

$$S = \frac{1}{r_s + \frac{r_p}{n}}$$

ecuación 2.5.1: Ley de Amdahl.

Donde S es el aumento de rendimiento, n es el número de procesadores en que se reparte la tarea, r_s representa la parte del problema que es secuencial y r_p la parte paralelizable, siendo $r_s + r_p = 1$.

La conclusión de este hecho es que la paralelización solo reporta ventajas significativas en problemas en los que la parte secuencial ocupe una fracción pequeña del tiempo de ejecución.

2.5.3 Tipos de paralelización

Podemos distinguir cuatro tipos de paralelización:

- **Paralelización a nivel de bit:** consiste en aumentar el tamaño de palabra que maneja un procesador. De este modo pueden manejarse datos de mayor tamaño de una vez, en lugar de tener que realizar las operaciones por partes. Fue la tendencia dominante desde los 70 hasta 1986.[4]

- **Paralelización a nivel de instrucción**: Consiste en reducir el tiempo que tarda el procesador en realizar una serie de instrucciones, aprovechando distintas estrategias:

  -Pipelining: Dado que un procesador se compone de distintos módulos que se ocupan de las distintas funciones necesarias para ejecutar una instrucción, es posible que varias instrucciones se estén ejecutando a la vez, una en cada etapa.

  -Ejecución superescalar: Se denomina superescalar a un procesador que tiene la capacidad de ejecutar a la vez más de una instrucción en la misma etapa. El número de instrucciones simultáneas puede variar según la complejidad de las mismas.

  -Ejecución fuera de orden: Cuando el procesador se encuentra con una instrucción que no puede ejecutar por que aún no tiene las entradas necesarias, en lugar de esperar la deja en una cola, y va ejecutando otras que sí puede. Los resultados de estas se dejan en una cola de salida y cuando la que faltaba se puede hacer, su resultado se coloca el primero y se envía la cola como si se hubiera ejecutado en su orden original.

  -Renombrado de registros: Cuando el código emplea el mismo registro para operaciones independientes, obliga a que estas se ejecuten de forma secuencial. Para evitar esto, los compiladores, o el propio procesador renombran estos registros cuando es posible, de forma que las operaciones pueden ejecutarse paralelamente (Por ejemplo, mediante Pipelining o ejecución superescalar).

-Ejecución especulativa: Consiste en aprovechar recursos sobrantes del procesador, para ejecutar código que aún no es seguro que sea necesario ejecutar. Un ejemplo de aplicación es ejecutar el código dentro de una expresión condicional (por ejemplo un "if") a la vez que se evalúa la condición. Si la condición se cumple, el código ya está ejecutado, y si no, se desecha el resultado.

-Predicción de saltos: Cuando una condición produce un salto, las instrucciones en el pipeline del procesador deben desecharse. Para evitar el desperdicio que esto supone, el procesador intenta "predecir" los saltos, basándose en los resultados de los condicionales anteriores.

- **Paralelización a nivel de datos:** Cuando una operación tiene que realizarse sobre varios datos, es posible realizar la operación de forma simultánea para todos ellos. Por ejemplo, supongamos la suma de dos vectores, A y B, que se guarda en uno C. Es posible realizar $a_i + b_i = c_i$ para todos los elementos simultáneamente, ya que no existe relación entre las operaciones.

- **Paralelización a nivel de tarea**: Consiste en realizar de manera simultánea más de una tarea distinta, sin que tenga porque existir ningún tipo de relación entre ellas. Se diferencia del paralelismo a nivel de datos en que las instrucciones que se ejecutan a la vez pueden ser distintas.

2.5.4 Clasificación de las computadoras paralelas

Michael J. Flynn propuso en 1972 la siguiente clasificación de arquitecturas de ordenador, atendiendo al número de instrucciones y datos que son capaces de manejar simultáneamente[10]:

- **Una instrucción, un dato (SISD):** Ejecución completamente en serie.
- **Una instrucción, datos múltiples (SIMD):** Aplicación de las mismas instrucciones a un conjunto de datos.
- **Múltiples instrucciones, un dato(MISD):** Este tipo de ordenador tiene una aplicación muy limitada en la realidad. Un ejemplo puede ser máquinas a prueba de errores, que ejecuten varias veces el mismo cálculo.
- **Múltiples instrucciones, múltiples datos(MIMD):** La máquina tiene capacidad de ejecutar varias instrucciones a la vez sobre datos diferentes.

2.5.5 Tipos de computadoras paralelas

En la actualidad, existen muchos tipos de computadoras con mayor o menor grado de paralelismo, a continuación se listan las más relevantes:

- **Procesadores multinúcleo**: Un procesador multinúcleo incluye dentro de un mismo chip, varios núcleos capaces de ejecutar instrucciones distintas en el mismo ciclo de reloj. Según la taxonomía de Flynn, entrarían dentro de la categoría MIMD. Además, cada uno de los núcleos puede emplear otras técnicas de paralelización como la ejecución superescalar. Estos procesadores son la tendencia dominante en la actualidad en el ámbito de los ordenadores domésticos, desde que en año 2004 Intel abandonase el incremento de frecuencia como vía de mejora del rendimiento a favor del incremento del número de núcleos[5]. Antes de que los procesadores multinúcleo se extendieran, algunos equipos de alto rendimiento disponían de más de un procesador mononúcleo para lograr el mismo resultado. A este tipo de configuración se le denomina **multiprocesamiento simétrico**. En la **figura 2.5.2** vemos el interior de un procesador multinúcleo, con los distintos núcleos(*cores,* en inglés) señalados:

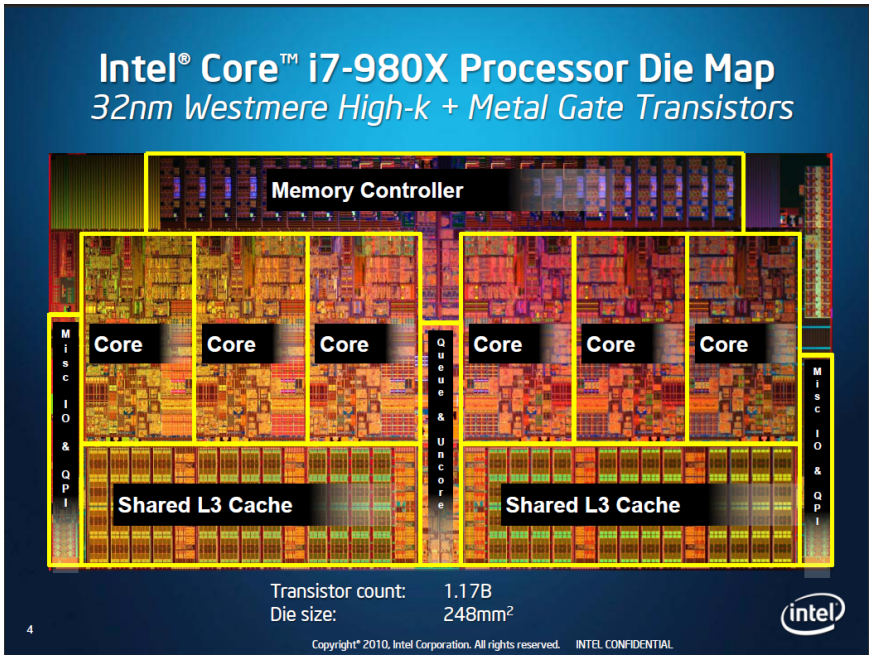

figura 2.5.2: Interior de un procesador multinúcleo[13].

- **Computación distribuida**: Varios ordenadores se conectan entre sí mediante una red, permitiendo distribuir entre ellos el trabajo a realizar. Un ejemplo de este tipo de configuración, es el proyecto SETI@home[6]. Este proyecto pide la colaboración de cualquiera que tenga un ordenador con acceso a internet para analizar datos de radiotelescopios, en busca de vida extraterrestre. En este caso, un servidor central envía los datos a los ordenadores de los voluntarios, que se encargan de su procesado.

- **Clúster**: Un clúster consiste en varios ordenadores conectados entre sí, mediante una red de alta velocidad, que de cara al usuario se comporta

como una sola máquina. Es un caso particular de computación distribuida, y es la configuración más utilizada en superordenadores. Es altamente escalable, ya que pueden añadirse y quitarse ordenadores a la red a voluntad. Admite tanto redes formadas por ordenadores idénticos como redes de ordenadores distintos. En la **figura 2.5.3** vemos un ejemplo de clúster, constituido por ordenadores de sobremesa convencionales, conectados entre sí.

figura 2.5.3: Ejemplo de clúster [2].

- **Procesador masivamente paralelo**: Una red de ordenadores, unidos por una conexión de alta velocidad especial. Se diferencia del clúster en que estos emplean redes estándar para conectar los equipos, mientras que en un procesador masivamente paralelo se diseñan redes especiales. Por este motivo, suelen ser siempre equipos muy grandes. En la **figura 2.5.4** podemos ver un ejemplo, el supercomputador Blue Gene/P:

figura 2.5.4: Supercomputador Blue Gene/P [2].

- **Computación paralela mediante FPGA**: Una FPGA (Field Programmable Gate Array) es un circuito integrado que puede configurarse para realizar diversas funciones lógicas. Antes de su uso deben configurarse mediante un lenguaje de descripción de hardware para realizar las operaciones necesarias. Conectadas a un ordenador, pueden emplearse como coprocesador, programadas para realizar un cálculo concreto. Dado que deben reconfigurarse para realizar cada operación, son adecuadas para tareas que requieran realizar los mismos cálculos una y otra vez, pero no para aplicaciones que requieran cálculos distintos constantemente. A continuación, en la **figura 2.5.5**, vemos un ejemplo de FPGA, que presenta una conexión PCIe, similar a la de una tarjeta gráfica o un adaptador de red.

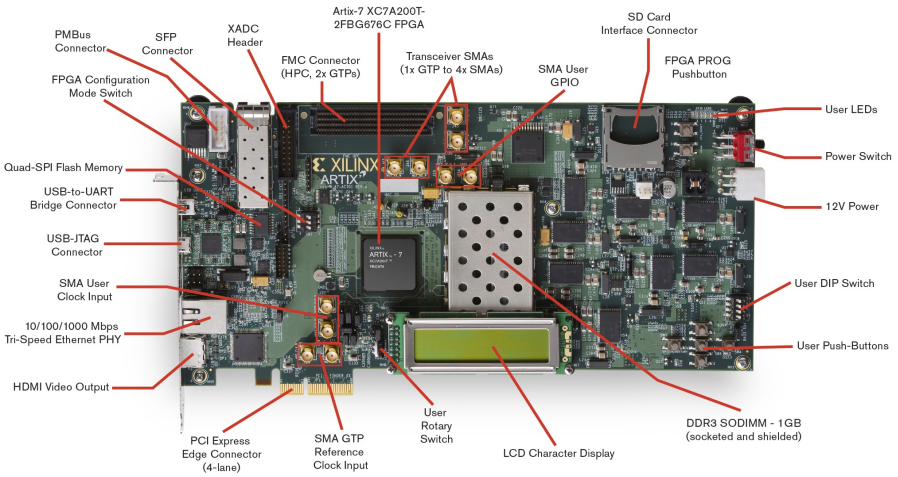

figura 2.5.5: Ejemplo de FPGA. Imagen propiedad del fabricante [14].

- **Computación paralela mediante ASIC**: Los ASIC son circuitos integrados hechos a medida para una aplicación concreta. Poseen las mismas capacidades que una FPGA, con mejor rendimiento, menor consumo y la posibilidad de realizar cálculos tan complejos como queramos. No obstante, la creación de un ASIC en un proceso tremendamente caro, y teniendo en cuenta el ritmo al que mejoran los procesadores de propósito general, esta inversión rara vez es rentable. Un ejemplo es el MDGRAPE-3 del instituto de investigación Riken en Japón, dedicado específicamente a la dinámica molecular. Este equipo combina más de 300 procesadores x86 comerciales con 4824 ASIC creados especialmente para él, lo que le permite lograr un rendimiento del orden de 10^{15} operaciones de coma flotante por segundo[7].

- **Procesadores vectoriales**: Se trata de procesadores que no operan con datos individuales, sino con vectores de datos. En la actualidad están en desuso, debido a lo rápido que evolucionaron los procesadores corrientes (escalares o superescalares).
- **Cómputo de propósito general en unidades de procesamiento gráfico(GPGPU):** Las unidades de procesamiento gráfico (GPU), son coprocesadores especializados en generar imágenes, liberando al procesador principal de esa tarea. Debido a la demanda constante de equipos más potentes, estas GPU se han convertido en coprocesadores tremendamente potentes y rápidos, con multitud de procesadores multihilo y anchos de banda altísimos[8]. En los últimos años, estos coprocesadores han comenzado a usarse para realizar cálculos no destinados a la generación de imágenes, dada su gran potencia y su alto nivel de paralelización. Existen varios lenguajes de programación destinados a GPGPU. Los dos más importantes son **OpenCL**, de código abierto y multiplataforma (Trabaja sobre GPU de NVIDIA y AMD, y también sobre CPU multinúcleo[9]) y **NVIDIA CUDA,** un lenguaje desarrollado por NVIDIA para ser usado sobre sus tarjetas gráficas y coprocesadores matemáticos dedicados. A continuación, en la **figura 2.5.6,** vemos un ejemplo de tarjeta gráfica, que además se corresponde al modelo empleado en este proyecto:

figura 2.5.6: Ejemplo de GPU [15].

- **Coprocesadores matemáticos:** Se trata de coprocesadores fabricados ex profeso para ejecutar cálculos de forma masivamente paralela. Pueden conectarse a un ordenador mediante una ranura PCIe, y suelen contar con versiones adaptadas a armarios de servidores y otras para torres convencionales. Cabe destacar los modelos **Tesla de NVIDIA**[11], basados en sus modelos más potentes de tarjetas gráficas, diferenciándose exteriormente por carecer de salidas de vídeo, pero con mayores

diferencias en el interior orientadas al uso que van a tener, como más memoria. Recientemente, Intel ha presentado sus propios coprocesadores, los **Xeon Phi**[12], basados en la arquitectura MIC (*Many integrated cores*, muchos núcleos integrados) Esta arquitectura consta de decenas de núcleos x86, similares a los Intel Xeon(la gama de servidores de Intel) lo que facilita la reutilización de código ya existente y, más importante, permite probar código de MIC en cualquier procesador Intel Xeon. Esto facilita el desarrollo de aplicaciones, ya que no es necesario disponer del equipo para probar el código. A continuación, en la **figura 2.5.7**, podemos ver un coprocesador Xeon Phi, citado anteriormente. Nótese que carece de ventilador, a diferencia de lo que suele ser habitual en tarjetas gráficas. Se debe a que este modelo concreto está diseñado para colocarse en servidores montados en armarios con sistemas de ventilación propio.

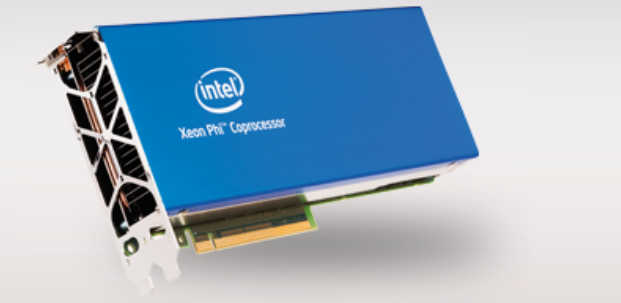

figura 2.5.7: Coprocesador Xeon Phi [12].

Referencias

[1] https://computing.llnl.gov/tutorials/parallel_comp/ Consultado el 18/08/2013.

[2] http://es.wikipedia.org/wiki/Computación_paralela Consultado el 18/08/2013.

[3] Amdahl, Gene (1967). "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities". AFIPS Conference Proceedings (30): 483–485. (Disponible en internet (a 18/08/2013) en http://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf).

[4] http://en.wikipedia.org/wiki/Bit-level_parallelism Consultado el 18/08/2013.

[5] http://www.nytimes.com/2004/05/08/business/08chip.html?ex=1399348800&en=98cc44ca97b1a562&ei=5007 Consultado el 19/08/2013.

[6] http://setiathome.ssl.berkeley.edu/ Consultado el 19/08/2013.

[7] http://www.rikenresearch.riken.jp/eng/roundup/4484.html Consultado el 19/08/2013.

[8]Introduction, From Graphics Processing to General Purpose Parallel Computing. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html Consultado el 15/08/2013.

[9] http://www.khronos.org/registry/cl/sdk/2.0/docs/man/xhtml/ Consultado el 15/08/2013.

[10] Flynn, M., Some Computer Organizations and Their Effectiveness, IEEE Trans. Comput., Vol. C-21, pp. 948, 1972

[11] http://www.nvidia.es/object/why-choose-tesla-es.html Consultado el 20/08/2013.

[12] http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html Consultado el 20/08/2013.

[13] www.intel.com

[14] www.xilinx.com

[15] www.gigabyte.com

2.6 CUDA

Bajo el nombre de CUDA, se engloba tanto la extensión del lenguaje C empleada para programar las GPU de NVIDIA, como el conjunto de herramientas desarrolladas para ello. En este apartado explicaremos, en primer lugar, las particularidades de la arquitectura de una GPU responsables de sus capacidades de cálculo y sus limitaciones y a continuación, la forma de programar en CUDA, atendiendo especialmente a la creación de hilos y a la copia de memoria entre la tarjeta y el equipo. Para finalizar, trataremos las herramientas de software proporcionadas por el fabricante, así como los archivos de código fuente y el tratamiento que les da el compilador.

2.6.1 Arquitectura de las GPU CUDA

Una GPU CUDA está dividida en multiprocesadores, cada uno de los cuales cuenta con varios núcleos para operaciones aritméticas(denominados por el fabricante núcleos CUDA, anteriormente conocidos como *shader units*), otros especiales para funciones trascendentes y uno o varios organizadores de *warps*(conjunto de 32 hilos, explicaremos este término en detalle más adelante).

Cabe mencionar que la arquitectura de las GPU es cambiante. A diferencia de lo que sucede con las CPU, en las que las arquitecturas son muy longevas. Con cada nueva arquitectura, los multiprocesadores se hacen más grandes e incorporan más prestaciones.

NVIDIA clasifica a sus tarjetas en función de sus prestaciones mediante la denominada "Capacidad de cómputo". Esta capacidad de cómputo se expresa mediante dos números, separados por un punto. El primero de ellos dependiente de la arquitectura de la GPU, es el que mayores cambios supone. El segundo número corresponde a pequeñas variaciones de la misma arquitectura.

Actualmente la capacidad más reciente es la 3.5, empleada en los coprocesadores Tesla y en los modelos de más alta gama de tarjetas gráficas. El equipo en el que se desarrolla este trabajo cuenta con una tarjeta de capacidad 3.0.

En estas dos versiones, los multiprocesadores están formados por los elementos siguientes: 192 núcleos CUDA, 32 núcleos para funciones trascendentes y 4 organizadores de *warps*.

Así mismo, cada GPU está compuesta de un número variable de multiprocesadores, lo que les permite multiplicar su rendimiento. La tarjeta empleada en el proyecto cuenta con dos multiprocesadores de capacidad 3.0, es decir, 384 núcleos CUDA.

Cada uno de estos multiprocesadores funciona bajo un esquema SIMD, de forma que todos sus núcleos ejecutan la misma instrucción a la vez.

Salta a vista la principal capacidad de esta arquitectura: ejecutar cientos, o miles de hilos que realicen las mismas operaciones sobre grandes conjuntos de datos.

2.6.2 Organización de los hilos en CUDA

A diferencia del código programado para CPU, en el que los hilos se lanzan de uno en uno, y son por norma general hilos muy pesados, en CUDA los hilos se lanzan por cientos, incluso por cientos de miles, pero suelen ser hilos mucho más ligeros. Para organizar estos hilos, la GPU emplea el siguiente esquema:

Para empezar, los hilos se agrupan en conjuntos de 32 denominados "*warp*". Este conjunto es la unidad mínima que manejará un multiprocesador. Pueden lanzarse grupos de menos de 32, pero los recursos que ocuparía el *warp* completo permanecerán vacíos. Esto es así por la estructura SIMD: El multiprocesador está hecho para enviar las mismas instrucciones a los 32 hilos del *warp* a la vez.

Existe un límite en el número de hilos que un multiprocesador puede manejar: 1024 en capacidad 3.0 y superior y 512 en el resto. Si queremos lanzar más de este número, es necesario dividir los hilos en bloques. Un bloque es el conjunto de hilos que entrará al multiprocesador al mismo tiempo. Este los dividirá en *warps* para ejecutarlos, pero todos los datos que manejen todos los hilos del bloque estarán disponibles siempre, independientemente del *warp* que se esté ejecutando. A efectos prácticos, puede considerarse que todos los hilos de un bloque se ejecutan simultáneamente, por tanto, unos hilos pueden hacer uso de los resultados de otros.

Los distintos bloques en que dividamos los hilos se agrupan en una malla. En el caso de que tengamos varios multiprocesadores, cada uno ejecutará un bloque. En el caso de que tengamos solo uno, o menos multiprocesadores que bloques, los bloques se ejecutarán secuencialmente. Este escalado se produce automáticamente, sin que deba ser programado.

Es importante comprender que, independientemente del número de multiprocesadores que tengamos disponibles, nunca se puede confiar en que los bloques se ejecuten simultáneamente o en un determinado orden.

Las dimensiones que puede tener esta malla son de 2^{31}-1 por 65535 bloques, es decir, 144.000 billones de hilos, lo que supone en la práctica que podremos lanzar cuantos hilos necesitemos.

2.6.3 Lenguaje de programación CUDA

El lenguaje de programación en que está basado CUDA es C/C++. Es posible emplear otros lenguajes mediante el uso de *wrappers*, o utilizar el código ensamblador de la propia tarjeta, denominado PTX(Solo para el código que ejecutará la tarjeta). Todo lo que se menciona aquí, hace referencia a la programación utilizando C/C++, presuponiendo un conocimiento adecuado del mismo.

Un programa de CUDA consta de código que se ejecuta en la CPU, como lo haría un programa convencional, y código que se ejecuta en la GPU. En la literatura, se utiliza el termino "*host*" para hacer referencia a la CPU y "*device*" para referirnos a la GPU. Se ha decidido mantener estos nombres, y alguno que aparecerá más adelante, por una parte, para familiarizar al lector con los términos que encontrará en cualquier documento sobre el tema y por otra, por que son palabras reservadas del lenguaje de programación, como ya veremos más adelante, por lo que también se facilita la lectura del código.

Los lanzamientos de hilos que se ejecutarán en la GPU se realizan desde el código host (el ejecutado por la CPU). Para lanzar conjunto de hilos, se emplea un tipo especial de función, denominado "*kernel*". Esta función, siempre de tipo *void*, se declara añadiéndole delante la expresión "__global__". Un ejemplo de declaración de *kernel* podría ser la siguiente:

```
__global__ void addKernel(int *c, const int *a, const int *b)
```

figura 2.6.1: Declaración de un *kernel* en CUDA.

A la hora de llamar a un *kernel*, existe una sintaxis especial, antes del nombre de la función se coloca la siguiente expresión: "<<< malla, bloques, flujo>>>" Los dos primeros valores corresponden, el primero, al número de bloques, y el segundo, al número de hilos por bloque. Ambos pueden ser, o bien un entero, o bien un elemento de un tipo especial, "dim3", que es básicamente un conjunto de tres enteros, correspondientes a los ejes X, Y y Z, para facilitar el trabajo con matrices y otras estructuras de datos de dos o tres dimensiones.

Respecto al tercer término, "flujo", es utilizado para lanzar varios grupos de hilos a la vez, ya que todos los *kernels* de un mismo flujo se ejecutan de forma secuencial. Si este campo se deja en blanco, se usará el flujo por defecto, que se crea automáticamente. A continuación vemos como se llamaría al *kernel* del ejemplo anterior, usando el flujo por defecto:

```
addKernel<<<1, size>>>(dev_c, dev_a, dev_b);
```

figura 2.6.2: Llamada a un *kernel* en CUDA.

El código del *kernel* será el que ejecuten todos los hilos. Para darle a cada hilo un trabajo distinto, CUDA incluye por defecto una serie de variables, accesibles desde dentro del *kernel*:

- **blockIdx**: Número de bloque al que pertenece el hilo.
- **blockDim**: Tamaño del bloque en que se encuentra el hilo.
- **threadIdx**: Número de hilo, dentro del bloque.

Estos tres valores no son enteros, sino de la clase "dim3", y poseen tres atributos, x, y, z, accesibles mediante la sintaxis habitual de las clases de C++ (blockIdx.x, por ejemplo). En el caso de que usáramos enteros a la hora de lanzar el *kernel*, las

coordenadas Y y Z de estos objetos tendrán siempre valor 0. Veamos un ejemplo del uso de estos valores:

```
__global__ void addKernel(int *c, const int *a, const int *b)
{
    int i = blockIdx.x*blockDim.x+threadIdx.x;
    c[i] = a[i] + b[i];
}
```

figura 2.6.3: Empleo de los identificadores de hilo para particularizar el trabajo.

Hay que tener en cuenta que la CPU Y la GPU utilizan distintas memorias RAM. Las variables que son usadas desde el *host*, deben estar en la memoria RAM del sistema, mientras que las variables que sean usadas por el código *device* tendrán que estar en la memoria de la GPU. Cuando lanzamos un *kernel*, los parámetros que le pasamos se copian a la memoria de la tarjeta.

Si Queremos emplear memoria dinámica, Existen dos funciones, análogas a las de C++ new y delete, con las que reservar y borrar memoria dinámica. Así mismo, mediante una tercera función podemos copiar el contenido de variables entre *host* y *device*. A continuación veremos un ejemplo de reserva de memoria dinámica, copia de *host* a *device*, ejecución de un *kernel* y copia del resultado de *device* a *host*.

```
//Creamos un puntero por variable
int *dev_a = 0;
int *dev_b = 0;
int *dev_c = 0;
int size; //(Tamaño de los vectores)


// Reservamos memoria para cada uno
cudaMalloc((void**)&dev_c, size * sizeof(int));
cudaMalloc((void**)&dev_a, size * sizeof(int));
cudaMalloc((void**)&dev_b, size * sizeof(int));

// Copiamos las entradas.
cudaMemcpy(dev_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, b, size * sizeof(int), cudaMemcpyHostToDevice);


// Lanzamos el kernel, con 1 bloque de "size" hilos
addKernel<<<1, size>>>(dev_c, dev_a, dev_b);

// Copiamos la salida a la memoria host.
cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);
```

figura 2.6.4: Ejemplo del uso de memoria dinámica en CUDA.

Una vez visto el funcionamiento de los *kernel* y la memoria, podemos comenzar a trabajar con CUDA. En la guía de programación disponible en [1] podemos

encontrar, entre otras cosas, un glosario completo con todas las funciones que podemos emplear en el código *device*. Las funciones que en C++ se encuentran en la librería estándar "math.h", se encuentran implementadas con el mismo nombre en CUDA, para facilitar su uso a programadores habituados a C++. Dado que estas funciones bastan para el trabajo a desarrollar, no entraremos a tratar otras librerías disponibles, enfocadas a temas específicos.

2.6.4 Herramientas de software

NVIDIA proporciona todas las herramientas necesarias para trabajar con CUDA a través de su página web [2]. El conjunto básico para Windows incluye el kit de herramientas CUDA (compilador, librerías, controladores…). Un conjunto de programas de ejemplo y la herramienta Nsight, un software de detección de errores (*debbuger*) que se integra con el entorno de desarrollo de Microsoft, Visual Studio. Además de servir de *debugger*, también añade CUDA como lenguaje a Visual Studio, de forma que podremos crear proyectos y añadir archivos como cualquier lenguaje de los incluidos por defecto.

En el caso de Windows, Nvidia proporciona soporte para Visual Studio, de forma que si contamos con él, bastará con ejecutar el instalador de CUDA y todo se configurará automáticamente. Es posible emplear otros entornos de desarrollo, o prescindir de ellos, utilizando el compilador de CUDA directamente.

En nuestro caso, emplearemos Visual Studio 2010 (Al comenzar el proyecto, Visual Studio 2012 no era compatible, aunque a día de hoy ya lo es).

Una vez instalado CUDA, ya podemos empezar a utilizarlo. Para ello, abriremos Visual Studio, y crearemos un nuevo proyecto, seleccionando como plantilla "CUDA", dentro de la categoría NVIDIA.

El proyecto comienza con un fichero con extensión ".cu" Estos archivos son el equivalente al ".cpp" de C++, archivos de código fuente, donde escribiremos nuestro código. Podemos añadir también archivos de encabezados (como los ".h" de C++), que en CUDA reciben la extensión ".cuh".

Nuestro proyecto puede contener archivos tanto de CUDA como de C/C++, si bien el código que use las librerías CUDA sólo puede aparecer en los archivos ".cu" y ".cuh". Si intentamos compilar un archivo corriente de C/C++ con código CUDA, recibiremos un error, ya que estos archivos son compilados por el compilador de C++ y no por el de CUDA.

Al crear el proyecto, vemos que el archivo por defecto ya incluye las bibliotecas básicas de CUDA, además de un programa de ejemplo, en el que se suman dos vectores, y que es un buen punto de partida para aprender el manejo básico de CUDA.

A la hora de compilar, nos encontramos con algunas particularidades:

- **Se emplean dos compiladores:** Visual Studio llama al compilador de CUDA para hacer el trabajo, pero este a su vez envía las partes de código que ejecutará la CPU al compilador de Visual Studio, y sólo se encarga del código de la GPU.
- **El código de la GPU no se guarda en formato binario:** Dadas las distintas arquitecturas de las GPU soportadas, los juegos de instrucciones son distintos, por lo que el código binario no funcionaría en todas. En su lugar, el código se deja en un estado intermedio, el código ensamblador de NVIDIA, PTX. Cada vez que un equipo va a emplear un programa nuevo que incluye código en CUDA, este es compilado justo antes de la ejecución con un compilador que incluyen todos los controladores de NVIDIA, aunque no esté instalado el juego de herramientas de desarrollo.
- **El código debe compilarse estableciendo una capacidad de cómputo mínima, y puede no funcionar en tarjetas de menos capacidad:** Cada revisión de CUDA añade características nuevas, pero muchas de ellas no pueden ser aplicadas a GPU anteriores, por limitaciones propias de su tecnología. Por ejemplo, las primeras versiones no soportaban variables de tipo "double". Si compilamos código que incluya variables de este tipo, no podrá usarse en ellas, sin embargo, si le indicamos al compilador que queremos código compatible con todas las tarjetas, sustituirá los "double" por "float". En otros casos, por ejemplo si llamamos a una función que no es soportada por la versión para la que compilamos, recibiremos un error.

Referencias

[1] http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html Consultado el 20/08/2013.

[2] https://developer.nvidia.com/cuda-downloads Consultado el 20/08/2013.

3. Alternativas de diseño

Tal como hemos visto anteriormente, existen múltiples alternativas en la actualidad para la computación paralela. A continuación vamos a compararlas, a fin de comprobar que la opción que emplearemos es la más adecuada.

Hemos de tener en cuenta que el objetivo es acelerar un algoritmo que ya se realiza en un ordenador estándar. Por tanto, la creación de un equipo a la medida para resolverlo(como un procesador masivamente paralelo, o un equipo basado en ASIC), resulta algo desproporcionado, ya que el coste económico sería desorbitado.

Podríamos plantear la creación de un clúster, empleando por ejemplo dos ordenadores corrientes. Suponiendo que los ordenadores sean idénticos, estaríamos hablando de un incremento del rendimiento que podría llegar idealmente al 100%, a costa de duplicar también el coste con respecto a un solo equipo.

Una opción interesante sería emplear un ordenador dotado de un procesador multinúcleo e implementar el algoritmo de forma que aproveche al máximo esta capacidad. Es una forma interesante de aumentar el rendimiento si hemos de realizar varias comparaciones, pero hacer que los distintos núcleos colaboren para realizar juntos la misma firma se antoja complicado, ya que parte del algoritmo exige que los hilos se sincronicen. Idealmente, podríamos aumentar el rendimiento hasta tantas veces como núcleos tenga el procesador. El coste, depende del modelo utilizado, como referencia tomaremos el que emplea el equipo de pruebas, que consta de 6 núcleos y cuesta en torno a los 100€ más IVA.

Las alternativas que nos quedan, pasan por utilizar un ordenador, ayudado por un sistema de coprocesado. Sin duda, la mayor potencia la obtendríamos de un coprocesador matemático especializado. Sin embargo, son unidades destinadas al mercado profesional, y cuentan con un precio bastante elevado, partiendo de 3000€ los modelos más baratos. El aumento de rendimiento podría ser enorme, pero dado que la tarea que va a realizar la realiza actualmente un ordenador corriente, seguramente sea una inversión desmesurada para el volumen de cálculo que realizará.

Existe una alternativa más barata a los coprocesadores matemáticos: el conocido como GPGPU, cálculo de propósito general en unidades de procesamiento gráfico. Esta técnica, convierte una tarjeta gráfica en un pequeño coprocesador matemático. Posee como gran ventaja su escalabilidad: podemos encontrar modelos desde poco más de 20€ hasta cerca de 1000€. Además, si elegimos un modelo del fabricante

NVIDIA, podemos emplear el mismo código tanto para GPGPU como para sus coprocesadores dedicados. Tomando un modelo de tarjeta de bajo precio (83€ más IVA), encontramos que dispone de 384 unidades de cálculo aritmético. Desde luego, supone una cifra muy interesante, y podría ofrecer una gran mejora con respecto al rendimiento del algoritmo en serie. Además, estaríamos adquiriendo una tarjeta gráfica que tiene muchas más aplicaciones además del cálculo paralelo, de forma que es muy factible que una vez terminado el estudio del algoritmo DTW, podamos amortizar la GPU en otros usos.

Como última alternativa, mencionaremos los sistemas basados en FPGA. Es posible encontrar modelos con un precio bastante reducido. como ejemplo citaremos la Avnet Spartan-6 LX9 MicroBoard[1], el modelo más básico de su fabricante, con un precio recomendado de 89USD (unos 65€) Sin embargo, atendiendo a sus características técnicas, vemos que cuenta con una memoria RAM bastante limitada (4,8Mb), lo que puede ser un problema para trabajar con grandes grupos de datos (la GPU mencionada antes cuenta con 1GB). Observando otros modelos, vemos que la memoria disponible aumenta mucho más despacio que el precio, lo que hace de la FPGA una opción limitada.

De todas las opciones vistas, tres han resultado ser viables, en términos de eficacia y proporcionalidad al problema: emplear una implementación multihilo en un procesador multinúcleo, emplear una tarjeta gráfica o emplear una FPGA.

Los hilos que es capaz de ejecutar a la vez una GPU, son muchos más que los de una CPU multinúcleo, por lo que preferimos la GPU de entre estos dos. La FPGA presenta una cantidad de memoria bastante limitada, que nos hace decantarnos por la GPU como solución final.

Una vez elegida la GPU como alternativa, podemos optar, como ya hemos visto en el apartado de computación paralela, entre dos lenguajes de programación. OpenCL, de código abierto y disponible para los dos principales fabricantes, AMD y NVIDIA, y CUDA, lenguaje propio de NVIDIA.

Dado que en el caso de CUDA, es el propio fabricante el que desarrolla también el lenguaje de programación y contamos con un completo kit de desarrollo gratuito (véase el apartado sobre CUDA), nos decantamos por esta vía. Como contrapartida, nos vemos obligados a elegir una GPU NVIDIA, sin poder optar por los modelos de AMD, aunque si comparamos la relación precio/prestaciones de ambos, no hay gran diferencia (Recordemos que se trata de productos destinados al público general).

4. Paralelización de operaciones

4.1 Paralelización del algoritmo DTW

Como hemos visto anteriormente, el algoritmo DTW se compone de dos pasos fundamentales: Calcular los costes locales y calcular los costes acumulados. Ambos procesos presentan un gran coste computacional, sin embargo, poseen una gran capacidad de paralelización, como veremos a continuación.

4.1.1 Cálculo de la matriz de costes locales

La matriz de costes locales C_{nxm} representa la distancia entre los puntos de los vectores A_n y B_m que estamos comparando, de forma que el elemento $c_{i,j}$ representa la distancia entre los puntos a_i y b_j.

En nuestro caso, utilizamos la distancia euclídea, que se calcula mediante la expresión de la **ecuación 4.1**:

$$D(a,b) = \sqrt{\sum_{k=0}^{n-1} (b_k - a_k)^2}$$

ecuación 4.1: Distancia euclídea entre dos puntos a y b n-dimensionales.

De esta forma, dado que trabajamos con puntos de cuatro dimensiones, cada punto de nuestra matriz quedará definido por la expresión de la **ecuación 4.2**, donde i y j representan los elementos de la matriz y los vectores y k representa cada una de las dimensiones de los puntos de A y B (en nuestro caso, emplearemos 4):

$$c_{i,j} = \sqrt{\sum_{k=0}^{4} \left(b_{j,k} - a_{i,k}\right)^2}$$

ecuación 4.2: Valor de un elemento genérico de la matriz de costes locales.

Como acabamos de ver, el cálculo de cada elemento de C no depende de los demás, por lo que **pueden realizarse todos en paralelo**. Secuencialmente, habría que realizar n·m cálculos.

4.1.2 Cálculo de la matriz de costes totales

El cálculo de los costes totales, o acumulados, es más complejo. Estos costes, como puede verse en [1], se definen como muestra la tabla de la **figura 4.3**, donde $d_{i,j}$, representa cada elemento de la matriz de costes totales, D. El elemento $d_{1,1}$ es simplemente $c_{1,1}$.

Primera fila	$d_{i,1} = \sum_{k=1}^{i} c_{k,1}$
Primera columna	$d_{1,j} = \sum_{k=1}^{j} c_{1,k}$
Resto de elementos	$d_{i,j} = c_{i,j} + \min\left(d_{i-1,j}, d_{i,j-1}, d_{i-1,j-1}\right)$

figura 4.3: Expresiones de los elementos de la matriz de coste acumulado.

Nótese que en el primer y segundo caso, la distancia puede reescribirse como se ve en la **ecuación 4.4**:

$$d_{i,1} = \sum_{k=1}^{i} c_{k,1} = c_{i,1} + \sum_{k=1}^{(i-1)} c_{k,1} = c_{i,1} + d_{(i-1),1}$$

$$d_{1,j} = \sum_{k=1}^{j} c_{1,k} = c_{1,j} + \sum_{k=1}^{(j-1)} c_{1,j} = c_{1,j} + d_{1,(j-1)}$$

ecuación 4.4: Desarrollo de las expresiones de los elementos de la primera fila y la primera columna de la matriz de costes acumulados.

En este caso sí existe dependencia entre los cálculos, por lo que la paralelización será limitada. Pasamos a examinar los distintos casos:

Comenzando por lo más evidente, cada elemento de la primera fila y de la primera columna depende del anterior, y solo del anterior. Por tanto, ambas deben realizarse de forma secuencial. Sin embargo, como los elementos de una no dependen de los de la otra, ambas pueden realizarse simultáneamente.

De la definición del resto de elementos, vemos que cada uno depende de tres: el que tiene debajo, el que tiene a la izquierda y el que tiene en su esquina inferior izquierda, como se ve en la **figura 4.5**:

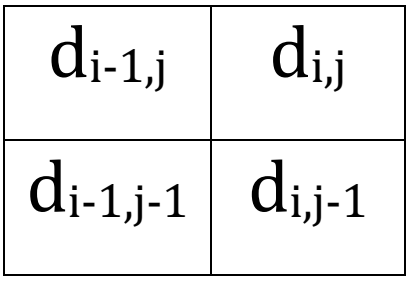

figura 4.5: Dependencias del elemento i,j de la matriz de costes totales.

De esto obtenemos dos conclusiones. La primera, es que es necesario calcular la primera fila y la primera columna para calcular los elementos centrales. La segunda, es que todos los elementos de los que depende $d_{i,j,}$ llamémoslos $d_{x,y}$, cumplen que $x+y<i+j$.

De esta segunda conclusión, se obtiene que si tomamos todos los elementos $d_{i,j}$ tales que $i+j=k$, podemos asegurar que son independientes. Cada uno de estos conjuntos de elementos corresponde a una diagonal de la matriz, tal como muestra la **figura 4.6:**

i+j	1,8	2,8	3,8	4,8	5,8	6,8	7,8	8,8
8	1,7	2,7	3,7	4,7	5,7	6,7	7,7	8,7
7	1,6	2,6	3,6	4,6	5,6	6,6	7,6	8,6
6	1,5	2,5	3,5	4,5	5,5	6,5	7,5	8,5
5	1,4	2,4	3,4	4,4	5,4	6,4	7,4	8,4
4	1,3	2,3	3,3	4,3	5,3	6,3	7,3	8,3
3	1,2	2,2	3,2	4,2	5,2	6,2	7,2	8,2
2	1,1	2,1	3,1	4,1	5,1	6,1	7,1	8,1

figura 4.6: Matriz de 8x8 con algunos de sus elementos de igual i+j resaltados.

Dado que los elementos de una misma diagonal son independientes entre sí, podremos ejecutarlos todos a la vez. De hecho, incluso los elementos de la primera fila y la primera columna presentan esta misma relación de dependencia, pero dado que la forma en la que se calculan es diferente y vamos a trabajar sobre un sistema SIMD, es preferible dejarlos fuera, aunque esto ya se tratará en profundidad más adelante.

La primera diagonal que realizaremos será i+j=4, teniendo en cuenta que excluimos la primera fila y la primera columna. La última será la que corresponda al elemento $d_{n,m}$, esto es i+j=n+m. Por tanto, **el número de diagonales a realizar secuencialmente es n+m-4**. Esto es un progreso considerable, teniendo en cuenta que de forma secuencial el número de pasos a realizar es n·m.

5. Desarrollo del software

5.1 Introducción

Una vez explicado el algoritmo DTW, el reconocimiento dinámico de firmas, la base de datos empleada y las herramientas y el lenguaje de CUDA, tenemos todo lo necesario para desarrollar el objetivo final del proyecto: una implementación paralela del algoritmo DTW, aplicada al reconocimiento de firmas.

Comenzaremos por extraer los datos de las firmas, para lo que será necesario desarrollar una aplicación ad hoc, partiendo de una escrita en instrucciones de MATLAB, que emplearemos como guía. También será necesario operar los datos obtenidos, para tomar las posiciones respecto al punto inicial de cada firma, y para obtener la velocidad del lápiz a partir de la velocidad.

Para procesar los datos, hemos de implementar el algoritmo DTW, cosa que haremos de dos formas: en paralelo, mediante CUDA, y en serie, mediante código C/C++ corriente. De esta forma podremos evaluar la diferencia de rendimiento entre un método y otro.

Precisamente para evaluar el rendimiento, utilizaremos una serie de herramientas que tanto CUDA como la biblioteca "Windows.h" ponen a nuestra disposición para cronometrar con precisión nuestro programa.

5.2 Extracción y tratamiento de datos

Al comenzar el proyecto se disponía de la base de datos de firmas, un programa para representarlas gráficamente (solo el ejecutable, no el código fuente) y un fichero ".m", con una función escrita en lenguaje MATLAB para abrir los archivos de la base de datos. Dado que no se encontró más información referente al formato de archivo utilizado (".fpg"), se optó por traducir el programa escrito para MATLAB a código C/C++.

El principal problema que se encontró fue el hecho de que MATLAB permite leer valores enteros de 1 o 2 bytes fácilmente, mientras que en C, el tamaño estándar es de 4 bytes, y para leer números menores hubo que leerlos como caracteres y posteriormente convertirlos.

Durante el proceso de traducción, se observó que muchos de los datos de la cabecera del fichero eran extraídos pero nunca usados (probablemente destinados a otras aplicaciones del formato de archivo), por lo que simplemente se saltaron.

Se decidió dividir la función original en dos: una para leer la cabecera del archivo, verificar que es correcto y obtener el tamaño que tienen las series de datos que forman la firma, y otra para extraer y almacenar los datos. De esta forma, sólo se reserva memoria si el archivo es correcto, además, de esta forma impedimos que se comparen firmas vacías.

Para comprobar que la apertura se realiza correctamente, se creó un programa de prueba, con el que mostramos los datos por pantalla y, dado que disponemos de un programa para visualizar las firmas gráficamente, podemos comparar los valores numéricos obtenidos con los de las gráficas, y comprobar que son correctos.

Una vez disponemos de un modo de leer los datos, y hemos comprobado que funciona, es necesario implementar una forma sencilla de manejar esta apertura, así como de generar los datos procesados a partir de los extraídos, y manejar todo ello de forma sencilla. Para ello se decidió crear una clase de C++: la clase firma.

La clase firma posee como atributos una serie de punteros, cada uno de los cuales se corresponde a una de las series de datos de la firma: x, y(posiciones), dx, dy(velocidades), p(presión),az(azimut) e in(inclinación). Además, también se almacena el tamaño de las series, en el atributo "size", de tipo entero. También cuenta con un constructor, en el que es necesario introducir la ruta de la firma que queremos abrir. Esto puede hacerse de dos formas: bien mediante la ruta completa del archivo, o bien, de forma más sencilla, introduciendo la ruta de la carpeta en la que se encuentra la base de datos e indicando la firma concreta que quiere abrirse, mediante tres variables: persona, firma, y verdad(que indica si queremos la firma verdadera o la falsa).

Además de cargar los datos, el propio constructor se encarga de procesarlos, referenciando las posiciones al primer punto capturado y obteniendo la velocidad a partir de las posiciones.

De este modo, simplemente creando un objeto de tipo "firma", ya disponemos de todos los datos listos para su uso.

5.3 Implementación del algoritmo DTW

A continuación se explican las implementaciones del algoritmo, haciendo hincapié en la implementación paralela, ya que es el objetivo de este trabajo. Como ya se ha explicado, el algoritmo consta de dos pasos fundamentales: cálculo de costes locales y cálculo de costes totales.

En la práctica esto se traduce en 4 *kernels* (funciones a ejecutar en la GPU) distintos, ya que, como se vio cuando estudiamos la paralelización del algoritmo, la expresión de los elementos de la matriz de costes totales tiene tres expresiones distintas. Dado que el tipo de paralelismo de CUDA es SIMD, lanzar los hilos con un único *kernel* provocaría que unos se ejecutasen mientras otros esperan, al tener que ejecutar instrucciones distintas. Por este motivo, la matriz de costes totales se calcula con 3 *kernels* distintos, que veremos más adelante.

Empezaremos por la matriz de costes locales, la más sencilla de calcular y también la primera que se necesita. Como todos los elementos de ella tienen la misma expresión, empleamos el mismo *kernel* para calcularlos todos. Asimismo, Dado que todos los elementos son independientes, podemos lanzar un hilo por cada elemento, de forma que se calcularán aprovechando al máximo la capacidad de la tarjeta gráfica.

Este *kernel* debe, por tanto, contener la expresión de un elemento genérico de la matriz de costes locales, que como se explica en la paralelización del algoritmo DTW es la siguiente:

$$c_{i,j} = \sqrt{\sum_{k=0}^{4} \left(b_{j,k} - a_{i,k}\right)^2}$$

ecuación 5.1: Expresión de un elemento genérico de la matriz de costes locales.

Es trivial caracterizar los elementos de la matriz, ya que como vimos anteriormente, cada hilo posee un identificador de 3 dimensiones (de las que en este caso usamos solo dos) tanto del bloque al que pertenece como del propio hilo dentro del bloque. Basta con asignar la coordenada “x” de los identificadores a la coordenada “i” de C y la coordenada “y” a “j”.

Los parámetros que necesita el *kernel* para ejecutarse son: los punteros a los vectores de entrada y a la matriz de salida, la dimensión de los puntos de los vectores y el tamaño de los vectores.

El tamaño de los vectores es necesario por dos motivos: el primero, por que la matriz C se almacena como un vector, y por tanto para acceder a sus elementos es necesario multiplicar el número de fila por su tamaño y sumar el número de columna.

El segundo, es una comprobación de seguridad: Antes de realizar el cálculo se comprueba que las coordenadas "i" y "j" son válidas. Esto es muy importante, ya que de no hacerlo podemos escribir en una posición incorrecta, pero válida, y no recibir ningún mensaje de error. (Por ejemplo, el elemento n+1 de la fila 2, es el elemento 1 de la fila 3).

Además, esto simplifica el lanzamiento de hilos, ya que al lanzar un *kernel*, los bloques deben ser del mismo tamaño, y si tenemos una matriz con un número de elementos no divisible entre el número de bloques necesarios, esto no es posible. De este modo, redondeamos al número más próximo y lanzamos un bloque al que seguramente le sobre una o dos filas o columnas, que simplemente no se ejecutarán al no cumplir la condición.

Esto no sólo simplifica el lanzamiento, también evita el tiempo que supone lanzar un segundo *kernel*, que no es despreciable.

Pasamos ahora al cálculo de la matriz de costes totales. En primer lugar se expondrá el caso más sencillo: la primera fila y la primera columna. Posteriormente se calculará el resto de la matriz.

La primera fila y la primera columna, como ya vimos, han de realizarse completamente en serie. En esta tarea sería mas rápido emplear la CPU, pues posee una velocidad mayor. Sin embargo, las copias de memoria entre la RAM del procesador y la RAM de la GPU suponen una gran perdida de tiempo, que hace recomendable emplear la GPU para realizar estos cálculos también.

Es necesario pues emplear dos *kernels* para estos cálculos. Podría haberse empleado uno solo, pero dado que la matriz no tiene por que ser cuadrada, se

prefirió dividirlo en dos. Además, dado que estos bucles tendrán solo un hilo cada uno, podrán ejecutarse simultáneamente en la mayoría de tarjetas gráficas.

La estructura de ambos es muy similar: constan de un bucle que va recorriendo la fila o la columna, sumando a cada elemento el coste acumulado del anterior.

Pasamos ahora a calcular el resto de los costes centrales. Cuando estudiamos las dependencias entre los elementos de la matriz, llegamos a la conclusión de que los elementos en una misma diagonal "i + j = k" eran independientes. Se procederá, por tanto, a calcular los elementos de cada diagonal de forma paralela. Existen dos opciones: la primera dado que el número de elementos de las diagonales es variable (creciente al principio y decreciente al final), parece interesante lanzar un *kernel* para cada diagonal, ajustando el número de hilos a la medida de esta. La segunda, lanzar un *kernel* con el número máximo de hilos, que recorra toda la matriz, operando sólo los que sea necesario cada vez.

Puede parecer que la primera opción es la más rentable, al eliminar los hilos innecesarios. Sin embargo, en la práctica se observó que el lanzamiento de un *kernel* cuesta tiempo, y lanzar tantos como diagonales tiene la matriz(m+n), es mucho más costoso que asumir que algunos hilos en algunas diagonales no hagan nada. Por este motivo se optó por incluir la segunda opción en el código final.

Para recorrer la matriz, será necesario caracterizar sus elementos en forma de dos variables: una que identifique la diagonal que estamos calculando, y otra que identifique el elemento dentro de esa diagonal.

Como dijimos antes, las diagonales son aquellas cuyos elementos cumplen "i + j = k ". Si despejamos j, tenemos= " j = k – i ". Si partimos de la diagonal i+j=n, podemos observar que a medida que retrocedemos, k disminuye, y a medida que avanzamos, k crece. Si tomamos i igual al número de hilo, podemos expresar j en función de i y de k. Para obtener el rango de k, observamos las diagonales que tenemos por delante y por detrás de la de referencia, tal como muestra la **figura 5.2:**

	7,0	7,1	7,2	7,3	7,4	k=(n-1)+(m-1)
	6,0	6,1	6,2	6,3	6,4	
k=n	5,0	5,1	5,2	5,3	5,4	
k=n-1	4,0	4,1	4,2	4,3	4,4	
	3,0	3,1	3,2	3,3	3,4	
k=2	2,0	2,1	2,2	2,3	2,4	
	1,0	1,1	1,2	1,3	1,4	
	0,0	0,1	0,2	0,3	0,4	

figura 5.2: Ejemplo a pequeña escala de la evolución de k.

Como se ve en la **figura 5.2**, los dado que el primer elemento a calcular es el [1,1] (la primera fila y la primera columna ya han sido calculadas), hemos de partir de k=i+j=1+1=2. Si vamos al extremo opuesto, el último elemento es [n-1,m-1], entonces k=i+j=n+m-2. Obtenemos aquí de nuevo lo que obtuvimos al discutir la paralelización del algoritmo= hemos de hacer n+m diagonales.

Como ya se ha dicho, se lanzarán n hilos, que ejecutarán un bucle n+m veces, incrementando k, dado que hay diagonales con menos de n elementos, verificaremos que al hilo le corresponde un elemento válido antes de calcular el coste acumulado.

Nótese en el ejemplo anterior que la matriz no es cuadrada. Esto no afecta a la forma de recorrer la matriz, es válida tanto para matrices cuadradas como para las que no lo son.

Una vez contamos con los 4 *kernels*, procedemos a unirlos en una sola función, con el objetivo de simplificar su uso. La función final, llamada costesCuda, recibe los vectores y una matriz de destino (ya en memoria de la tarjeta), y ejecuta el algoritmo. Así mismo, también devuelve el tiempo empleado, y si ha encontrado algún error.

5.4 Funciones para evaluación de rendimiento

Para evaluar el tiempo que una función tarda en ejecutarse en un entorno Windows, se dispone de unas funciones específicas en la biblioteca "winbase.h". CUDA, por su parte, posee sus propias funciones, que tienen como particularidad que sólo miden el tiempo cuando la GPU está participando en el programa.

Para la evaluación del algoritmo en paralelo, así como las copias a la memoria de la tarjeta, se han empleado las funciones de CUDA, que cuentan con la ventaja de ser multiplataforma, lo que facilitará la reutilización del código.

Sin embargo, al intentar usar los temporizadores de CUDA con la implementación en serie, se comprobó que, efectivamente, no marcaban nada. Se procedió entonces al uso de las funciones de la biblioteca "winbase.h", para poder medir el algoritmo en serie, aunque si se quisiera trasladar el código a otros sistemas operativos, habría que sustituirlas.

5.5 Desarrollo del programa final

Una vez explicadas las diferentes partes, procedemos a componer el programa final.

El programa se desarrolla de la siguiente manera: primero, se abren los ficheros que contienen las firmas, y se almacenan en memoria dinámica. Recordemos que la clase "firma" se ocupa de acondicionar los datos automáticamente. A continuación, los datos escogidos para calcular el algoritmo, son copiados a una serie de vectores, tanto en memoria *host*(la que usa la cpu) como *device* (la que usa la GPU).

Una vez tenemos todos los datos disponibles, ejecutamos el algoritmo de la forma que prefiramos, en serie o en paralelo. Para ello, reservamos una matriz de n x m en la memoria adecuada y llamamos a la función correspondiente. Esta función devuelve el tiempo que ha tardado en ejecutarse y la matriz de costes acumulados, de la que tomamos el elemento [n , m]. A continuación liberamos la memoria de la matriz.

El proceso puede repetirse las veces que queramos, mediante los bucles que ejecutan el algoritmo.

Además del tiempo que emplea el algoritmo, también existen temporizadores que miden los procesos de copia de memoria y de borrado, para disponer de datos sobre cada pieza del proceso.

6. Presupuesto

A continuación se detalla el presupuesto detallado del presente trabajo. Para su realización, se ha empleado la plantilla propuesta por la Universidad Carlos III, disponible en [1].

Como costes de personal, se cuentan tanto el correspondiente al autor del trabajo, como ingeniero, como los del tutor del proyecto, en calidad de ingeniero sénior.

En cuanto a los costes de amortización, se apuntan dos conceptos:

- **Equipo informático de pruebas:** Es el equipo detallado en el apartado "Rendimiento", en el que se realiza la programación del algoritmo y todas las pruebas realizadas.
- **Equipo informático auxiliar:** Se trata de un ordenador portátil, en concreto un Apple MacBook Pro. Este equipo es el ordenador personal del autor, y ha sido utilizado para la redacción de la memoria, así como para consultar recursos diversos durante la realización del proyecto.

En el apartado de "Otros gastos directos", se mencionan varios productos de software con coste cero. Esto se debe al acuerdo de la Universidad Carlos III con el fabricante Microsoft, que permite a los alumnos acceder a multitud de programas de dicho fabricante sin coste alguno, siempre que se utilicen para fines académicos.

Dado que este proyecto se basa en la utilización de la tarjeta gráfica para la ejecución del algoritmo, indicamos que el coste del modelo utilizado es de 96,78€(79,98€ sin IVA).

PRESUPUESTO DE PROYECTO

1.- Autor:

Alejandro García Molina

2.- Departamento:

Tecnología electrónica

3.- Descripción del Proyecto:

- Titulo: ACELERACIÓN CON GPU DE ALGORITMOS DE RECONOCIMIENTO BIOMÉTRICO MEDIANTE FIRMA MANUSCRITA ON-LINE
- Duración (meses): 6

Tasa de costes Indirectos: 20%

4.- Presupuesto total del Proyecto (valores en Euros):

Euros

5.- Desglose presupuestario (costes directos)

PERSONAL

Apellidos y nombre	N.I.F. (no rellenar - solo a titulo informativo)	Categoría	Dedicación (hombres mes) [a]	Coste hombre mes	Coste (Euro)	Firma de conformidad
					0,00	
Luis Mengíbar Pozo		Ingeniero Senior	0,091428571	4.289,54	392,19	
Alejandro García Molina		Ingeniero	2,285714286	2.694,39	6.158,61	
					0,00	
					0,00	
		Hombres mes	**2,377142857**	**Total**	6.550,79	

[a] 1 Hombre mes = 131,25 horas. Máximo anual de dedicación de 12 hombres mes (1575 horas)
Máximo anual para PDI de la Universidad Carlos III de Madrid de 8,8 hombres mes (1.155 horas)

EQUIPOS

Descripción	Coste (Euro)	% Uso dedicado proyecto	Dedicación (meses)	Periodo de depreciación	Coste imputable [d]
Equipo informático de pruebas	496,00	100	6	60	49,60
Equipo informático auxiliar	1.832,00	10	6	60	18,32
		100		60	0,00
		100		60	0,00
		100		60	0,00
					0,00
				Total	67,92

[d] Fórmula de cálculo de la Amortización:

$$\frac{A}{B} \text{xCxD}$$

A = nº de meses desde la fecha de facturación en que el equipo es utilizado
B = periodo de depreciación (60 meses)
C = coste del equipo (sin IVA)
D = % del uso que se dedica al proyecto (habitualmente 100%)

SUBCONTRATACIÓN DE TAREAS

Descripción	Empresa	Coste imputable
	Total	0,00

OTROS COSTES DIRECTOS DEL PROYECTO[e]

Descripción	Empresa	Costes imputable
Sistema operativo Windows 8	Microsoft	0,00
Visual Studio Ultimate 2010	Microsoft	0,00
	Total	0,00

[e] Este capítulo de gastos incluye todos los gastos no contemplados en los conceptos anteriores, por ejemplo: fungible, viajes y dietas, otros,…

6.- Resumen de costes

Presupuesto Costes Totales	**Presupuesto Costes Totales**
Personal	6.551
Amortización	68
Subcontratación de tareas	0
Costes de funcionamiento	0
Costes Indirectos	1.324
Total	7.942

Referencias

[1]http://www.uc3m.es/portal/page/portal/administracion_campus_leganes_est_cg/proyecto_fin_carrera /Formulario_PresupuestoPFC-TFG%20(3)_2.xlsx

7. Análisis de rendimiento

7.1 Introducción

Una vez el desarrollo del software ha concluido, podemos comprobar si la reducción de tiempo que teóricamente se obtiene de la paralelización del algoritmo es apreciable en la realidad.

Para esta prueba, se ha utilizado un procesador AMD 6300, con una frecuencia de reloj de 3,5 GHz. Consta de 6 núcleos, aunque dado que se emplea para evaluar el rendimiento del algoritmo en serie, no se ha implementado una solución que saque partido de esta característica. Como tarjeta gráfica, se ha empleado una NVIDIA GTX 650, con una frecuencia de reloj de 1089 MHz. Este modelo cuenta con dos multiprocesadores, que suman un total de 382 núcleos CUDA.

Para evaluar el rendimiento, se realizaron comprobaciones con todos sujetos de la base de datos (100 en total), dado que se observó una gran variación en el tamaño de las distintas series de datos, habiendo algunas de apenas 100 elementos y otras casi de 1000. Como teóricamente vimos que la complejidad del algoritmo en serie evoluciona con el cuadrado del tamaño y en paralelo linealmente, usar sólo un sujeto daría una estimación sesgada.

7.2 Medición de tiempos y extracción de datos

El proceso de comparar las firmas se compone de dos partes, las cuales se ejecutan un número distinto de veces: Por una parte, es necesario abrir los ficheros que contienen las firmas y, en el caso de CUDA, copiarlas desde la memoria general a la de la tarjeta gráfica. Este proceso se repite para cada firma que queramos comprobar. A continuación, procedemos a realizar las comparaciones propiamente dichas.

Ambos pasos presentan una duración distinta: la apertura en CUDA tiene un paso más, mientras que el algoritmo se desarrolla de forma distinta según sea en serie o en paralelo. Dado que, como ya hemos dicho, ambos pasos se ejecutan un número distinto de veces, mediremos su duración por separado.

En primer lugar, comparamos la apertura de archivos. Para ello, abrimos la base de datos completa, 5000 ficheros, con y sin la copia a la memoria de la tarjeta. Repetimos el proceso varias veces para tomar un valor medio. Posteriormente,

dividimos entre el número de archivos para obtener el tiempo medio por firma. Los datos obtenidos son los siguientes:

	CUDA	Serie
Tiempo de apertura total(ms)	2250	1455
Tiempo de apertura unitario(ms)	0,45	0,29

figura 7.1: Tiempos de apertura de los archivos de firma.

Como podemos ver en la **figura 7.1**, el tiempo que se tarda en abrir una firma se incrementa en un 50% en el caso de CUDA, debido a la copia de memoria necesaria.

En cuanto al algoritmo, procedemos de la siguiente manera: compararemos la primera firma verdadera de cada usuario con todas las demás, verdaderas y falsas, de ese usuario. Igualmente, repetimos las medidas para tomar una media. Al finalizar, dividimos el tiempo total entre el total de comparaciones realizadas y obtenemos el tiempo medio por comparación. Mostramos los resultados en la tabla de la **figura 7.2**:

	CUDA	Serie
Duración total del algoritmo(ms)	14410	30292
Duración media por comparación(ms)	3,00	6,31

figura 7.2: Comparativa entre la realización del algoritmo en serie y en paralelo.

En este paso tenemos un resultado opuesto: Como cabía esperar, la paralelización ofrece un resultado mejor, reduciendo a menos de la mitad la duración del algoritmo.

7.3 Evaluación de los datos obtenidos

Los resultados muestran una mejora en la duración del algoritmo, a costa de incrementar los tiempos de apertura. A continuación evaluaremos para qué casos resulta interesante emplear una solución paralela, teniendo en cuenta este hecho.

Podemos determinar la duración total de la comparación mediante la siguiente fórmula:

$$T = x \cdot t_{ap} + y \cdot t_{comp}$$

ecuación 7.3: Expresión de la duración del proceso de comparación

Donde T representa la duración total, t_{ap} y t_{comp} los tiempos de apertura y comparación, 'x' el número de archivos a abrir e 'y' el número de comparaciones a realizar.

De cara a la utilización práctica de este algoritmo, el número máximo de aperturas de archivos, será dos veces el número de comparaciones. Este caso correspondería a realizar comparaciones en las que ninguna firma se repite nunca. Este caso es, así mismo, el más desfavorable para la implementación en CUDA, ya que las aperturas son más lentas que en la implementación en serie. En este caso, podemos expresar el tiempo en función del número de firmas de la siguiente forma:

$$T = 2 \cdot y \cdot t_{ap} + y \cdot t_{comp}$$

ecuación 7.4: Expresión del tiempo total en el peor caso posible para CUDA.

En el extremo opuesto, tenemos el caso de realizar todas las comparaciones posibles para las firmas que abramos, incluyendo comparar cada firma consigo misma. En este caso, el número de comparaciones será el cuadrado del número de aperturas, es decir, el número de aperturas será la raíz cuadrada del número de comparaciones. Este caso es el más favorable para CUDA, pues minimiza su parte más lenta. En este caso, el tiempo total queda de la siguiente forma:

$$T = \sqrt{y} \cdot t_{ap} + y \cdot t_{comp}$$

ecuación 7.5: Expresión del tiempo total en el mejor escenario posible para CUDA.

Podemos considerar que, sea cual sea la relación entre el número de aperturas y el número de comparaciones, el tiempo total estará comprendido en algún punto entre el mejor y el peor caso posible. Por tanto, si representamos ambos gráficamente, obtendremos una horquilla dentro de la cual se situarán todos los casos posibles. Nótese que en el caso de la implementación en serie, el mejor y el peor caso es el contrario que en la implementación en CUDA, ya que en este caso las aperturas son más rápidas y las comparaciones más lentas.

Si comparamos las horquillas definidas por los casos límite de las implementaciones en serie y en paralelo, podemos evaluar de forma inmediata la aplicación más adecuada en cada caso, así como la ganancia de tiempo. Tomando los valores de tiempos de apertura y comparación unitarios obtenidos experimentalmente, obtenemos la representación gráfica que muestra la **figura 7.6:**

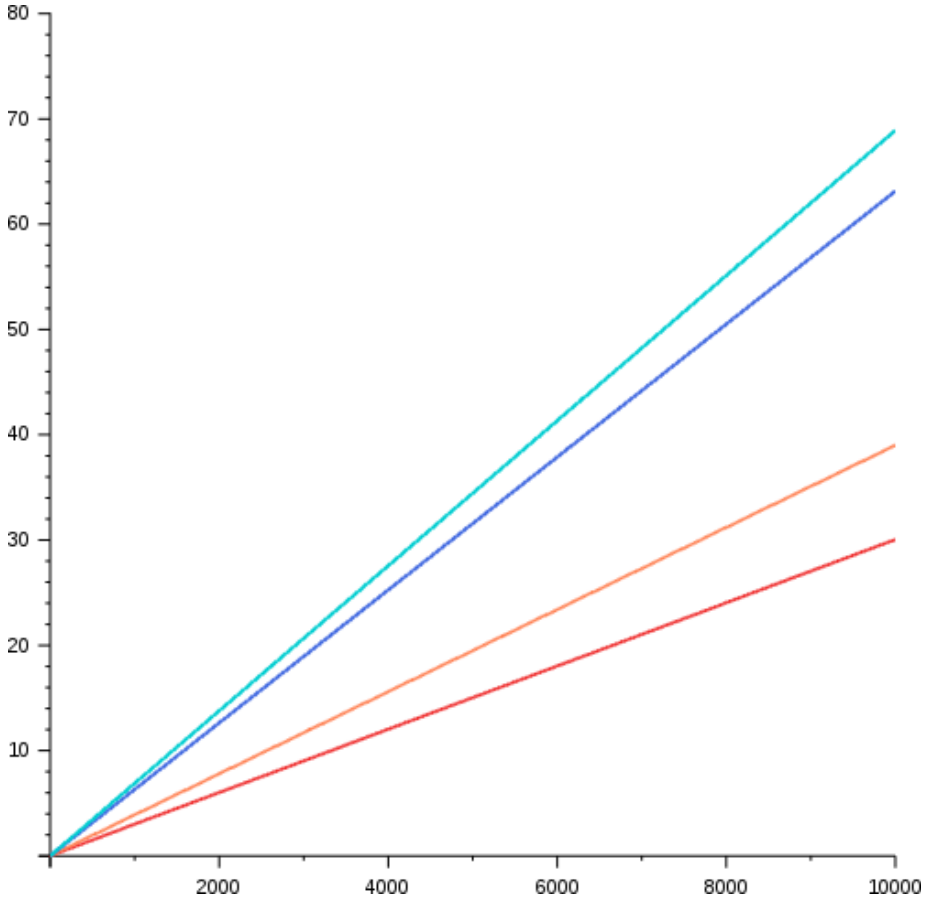

figura 7.6: Representación gráfica de la duración(en segundos) de las soluciones en serie(azules) y en paralelo(rojos), en función del número de comparaciones. El color claro corresponde al mejor caso de cada solución, y el color oscuro al peor (recordamos que el peor de uno es el mejor del otro).

Podemos ver que incluso en el peor escenario posible, la implementación en CUDA resulta más rápida. También vemos que, si suponemos un escenario intermedio, **el tiempo de la solución en paralelo es aproximadamente la mitad** que el de la solución en serie.

En vista de los resultados, podemos concluir que la utilización del algoritmo implementado de forma paralela resulta atractiva siempre que el tiempo empleado en ejecutar el algoritmo en serie resulte considerable.

7.4 Rendimiento en términos económicos

A continuación, analizaremos, desde un punto de vista económico, en qué momento resulta atractiva la implementación paralela del algoritmo DTW. Aunque nuestro trabajo versa sobre el reconocimiento de firmas, el algoritmo se desarrolla prácticamente de la misma forma para cualquier otra aplicación, por lo que las conclusiones serán igualmente válidas.

Situémonos en un marco académico o empresarial, poniéndonos en el papel de la institución. Nos planteamos la conveniencia de acelerar el algoritmo DTW, que un investigador, empleado nuestro, está utilizando en sus estudios. El motivo de plantearnos esto, es que el investigador no puede continuar con su trabajo mientras el algoritmo se ejecuta, ya que el ordenador está ocupado realizando cálculos y lo necesita para seguir con su tarea.

Nos encontramos pues con una ecuación sencilla: si el tiempo perdido se reduce, ahorraremos mano de obra, pero por otra parte, paralelizar el algoritmo conlleva una inversión inicial.

Pueden darse dos situaciones: que ya dispongamos de una GPU adecuada (algo nada extraño, ya que NVIDIA es uno de los principales fabricantes), o que debamos hacer la inversión económica de comprar una. Nótese que solo se considera el sobrecoste de la tarjeta, ya que el resto del equipo es necesario también para la implementación en serie.

Además, en cualquier caso, deberemos invertir una serie de horas de trabajo, bien de ese investigador, o bien de un programador informático, en implementar el algoritmo, instalar la tarjeta gráfica en el equipo, descargar las herramientas adecuadas, etc.

El coste de este desarrollo, sería aproximadamente el coste de este proyecto, detallado en el capítulo de presupuesto, el tiempo necesario para esta tarea es de 300 horas. Simplificadamente, asumiremos que el investigador, o el encargado de la implementación paralela, cobran por hora el mismo sueldo que aparece reflejado en el presupuesto.

En el caso de tener que adquirir la tarjeta, si optamos por el modelo empleado en este proyecto, realizaremos una inversión de 82,98€ (descontamos el IVA, por ser

una empresa, o una institución educativa). Esta inversión resulta prácticamente despreciable, si tenemos en cuenta el coste de desarrollo.

Si asumimos que nuestro investigador tiene el mismo sueldo que el empleado en el presupuesto para el ingeniero (20,5€/hora), y el coste total del proyecto es de 7942 €, obtenemos que debemos ahorrarnos 387 € horas para rentabilizar la inversión.

Estos resultados hacen muy difícil que este proyecto fuese rentable si sólo una persona va a utilizar el algoritmo. Recordamos además que las variaciones en las arquitecturas de NVIDIA puede dejar el código obsoleto en pocos años.

No obstante, hay que tener en cuenta que este proyecto ha sido realizado partiendo completamente desde cero, sin tener noción alguna de CUDA ni del algoritmo DTW. Sin embargo, esta no sería la situación real en el caso de una empresa: lo lógico sería buscar un programador que dominase el lenguaje CUDA.

En este caso, el propio desarrollo del algoritmo resulta bastante sencillo, y tal como muestra la planificación del trabajo, el algoritmo en sí ocupó aproximadamente 35 horas, que serían aproximadamente las que tardaríamos en amortizarlo.

En resumen, podemos estimar que la paralelización de este algoritmo resulta atractiva si puede ahorrarnos en torno a 35 horas de trabajo en el ámbito empresarial. Un caso distinto sería el de una institución educativa que pudiera desarrollar el algoritmo con un coste casi nulo, como por ejemplo mediante un proyecto de fin de carrera.

8. Conclusiones generales

Una vez completado el desarrollo y la evaluación de la implementación en CUDA del algoritmo DTW, nos hemos encontrado con una mejora del rendimiento con respecto a la implementación en serie de un 100%. Podemos pues considerar que hemos cumplido nuestro objetivo de acelerar el algoritmo. Sin embargo, considerando las alternativas que descartamos al principio, se plantea la duda de si otras alternativas hubieran sido más eficaces.

Según el apartado "Alternativas de diseño", otra opción a considerar hubiera sido el empleo de una implementación multihilo, ejecutada por un procesador multinúcleo. Pusimos como ejemplo un procesador de 6 núcleos, con el que teóricamente podríamos haber llegado a un 600% de incremento de rendimiento.

Desde un punto de vista teórico, esa alternativa parece mejor, sin embargo, hemos de considerar algunos puntos:

- CUDA ofrece gran facilidad de escalado, mientras que en un procesador multihilo, el escalado debe ser programado.
- La CPU no solo se ocuparía de ejecutar el algoritmo, también tendría que ocuparse del resto de tareas del ordenador (Sistema operativo, el entorno de desarrollo que estemos usando, otros programas ejecutándose en segundo plano, etc.)
- El uso de la GPU libera a la CPU de trabajo, permitiéndola dedicarse a otras tareas.
- Ambas opciones no son incompatibles: siempre podremos usar la implementación en CUDA junto con una implementación para CPU para trabajar con grandes volúmenes de datos.

Por tanto, se puede considerar el resultado del trabajo como satisfactorio: hemos descubierto hasta que punto es efectivo el GPGPU para trabajar con el algoritmo DTW y se ha acelerado la implementación de la que se partía.

También es importante tener en cuenta que gran parte de la carga de trabajo de este proyecto ha consistido en documentación y aprendizaje: Biometría, tipos de computación paralela, el algoritmo DTW y un nuevo lenguaje de programación, acompañado de una forma distinta de enfocar los problemas informáticos, en paralelo. Todo este aprendizaje es independiente de los resultados obtenidos, y constituye un añadido valioso para los conocimientos adquiridos durante el grado.

9. Trabajos futuros

Una vez completada la implementación mediante CUDA del algoritmo DTW, surgen varios caminos a seguir, explorando diversos aspectos tratados en el presente trabajo, buscando aplicaciones prácticas de las conclusiones obtenidas, o profundizando en ellas enfocándolas hacia casos concretos. Enumeramos los siguientes:

- **Implementación del algoritmo DTW mediante OpenGL:** OpenGL, como ya se mencionó en la sección de estado del arte, es una biblioteca de cálculo paralelo, similar a CUDA, pero no limitada a tarjetas gráficas NVIDIA, sino abierta a cualquier equipo capaz de realizar operaciones de forma paralela. Dado que esta librería está desarrollada por terceros, y teniendo en cuenta la gran complejidad que supone el constante cambio de arquitecturas de las GPU, puede resultar muy interesante comparar el rendimiento que es capaz de ofrecer frente a la solución del propio fabricante. Además, podría evaluarse también el rendimiento de otros dispositivos, como GPU de AMD, gran rival de NVIDIA, que no posee una biblioteca propietaria para GPGPU.

- **Implementación del algoritmo DTW en una FPGA:** Las FPGA suponen otra alternativa viable de coprocesador, con la desventaja de que deben ser configuradas para la tarea a realizar previamente. Si encontrásemos un incremento muy grande de rendimiento, no obstante, es posible que el tiempo invertido en su configuración se compense con una ejecución más rápida del algoritmo.

- **Combinar la implementación paralela con una en serie:** Las soluciones en serie y en paralelo no son mutuamente excluyentes: pueden desarrollarse de forma simultánea, ejecutando comparaciones distintas. Un programa capaz de distribuir el trabajo entre procesador y gráfica, en función del rendimiento que sea capaz de ofrecer cada uno, de forma que el conjunto de comparaciones a realizar consuma el menor tiempo posible, sería la opción más rápida que podría ofrecer un ordenador corriente.

- **Explorar las posibilidades de las GPU integradas:** Aunque de menor rendimiento en términos generales que las versiones dedicadas, las GPU integradas ofrecen una característica interesante: comparten memoria RAM con el procesador. Esto hace innecesaria la copia de datos entre memorias. Las posibilidades que esto ofrece no son sólo el omitir un paso en el desarrollo del algoritmo, si en el apartado anterior hablábamos de realizar comparaciones distintas en GPU y CPU, aquí podríamos ir un paso más lejos. Teniendo en cuenta la baja potencia de las GPU integradas, que además suelen ir emparejadas también con un procesador de gama baja, podrían emplearse los dos en la misma comparación, algo que no es posible en

equipos con memorias separadas, ya que habría que realizar cientos de copias de memoria. Quizá esto no tenga mucha relevancia en el caso concreto de las firmas, que forman series de datos bastante cortas, pero en otras aplicaciones que presenten series de miles de elementos, podría suponer mejoras importantes.

- **Desarrollo de una aplicación comercial para el bloqueo y desbloqueo de equipos con entrada táctil mediante firma:** En la actualidad, están surgiendo multitud de ordenadores domésticos, tanto en formato sobremesa como portátil o tableta, que cuentan con soporte táctil, no solo mediante los dedos, sino mediante punteros capaces de transmitir una posición precisa y la presión ejercida. En estos casos, el permitir desbloquear el equipo firmando en la pantalla puede tener una buena acogida por el usuario, ya que la firma resulta un medio familiar de identificación. Además, usar la GPU en lugar del procesador, libra a este de esa carga, permitiendo por ejemplo que si la identificación se realiza al encender el equipo, la CPU se centre en el arranque y deje a la GPU la tarea de identificar al usuario.

- **Adaptación de la implementación y reevaluación del rendimiento pasados unos años:** El ritmo al que se incrementan las prestaciones de las tarjetas gráficas, sumado al hecho ya mencionado de la volatilidad de sus arquitecturas, hace que las conclusiones de este proyecto tengan una validez temporal limitada. Como ejemplo, baste señalar que el mismo modelo de tarjeta empleado, de una sola generación menos (el GTX 550Ti, en lugar de GTX 650) poseía 192 núcleos divididos en 4 multiprocesadores, mientras que el modelo actual cuenta con 384, divididos en sólo 2 multiprocesadores.